This electronic version (PDF) was scanned by the International Telecommunication Union (ITU) Library & Archives Service from an original paper document in the ITU Library & Archives collections.

La présente version électronique (PDF) a été numérisée par le Service de la bibliothèque et des archives de l'Union internationale des télécommunications (UIT) à partir d'un document papier original des collections de ce service.

Esta versión electrónica (PDF) ha sido escaneada por el Servicio de Biblioteca y Archivos de la Unión Internacional de Telecomunicaciones (UIT) a partir de un documento impreso original de las colecciones del Servicio de Biblioteca y Archivos de la UIT.

هذه النسخة الإلكترونية (PDF) للاتصالات الدولي الاتحاد في والمحفوظات المكتبة قسم أجراه الضوئي بالمسح تصوير نتاج (ITU) والمحفوظات المكتبة قسم في المتوفرة الوثائق ضمن أصلية ورقية وثيقة من نقلاً.

此电子版（PDF版本）由国际电信联盟（ITU）图书馆和档案室利用存于该处的纸质文件扫描提供。

Настоящий электронный вариант (PDF) был подготовлен в библиотечно-архивной службе Международного союза электросвязи путем сканирования исходного документа в бумажной форме из библиотечно-архивной службы МСЭ.

© International Telecommunication Union

INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

YELLOW BOOK

VOLUME VI - FASCICLE VI.8

CCITT HIGH LEVEL LANGUAGE (CHILL)

RECOMMENDATION Z.200

VIITH PLENARY ASSEMBLY
GENEVA, 10-21 NOVEMBER 1980

Geneva 1981

INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

YELLOW BOOK

CORRIGENDUM TO FASCICLE VI.8

CCITT HIGH LEVEL LANGUAGE (CHILL)

First list of clerical errors in Recommendation Z.200

VIITH PLENARY ASSEMBLY
GENEVA, 10–21 NOVEMBER 1980

Geneva 1982

BIBLIOTHÈQUE DE L' UIT GENÈVE

CORRIGENDUM TO FASCICLE VI.8 OF THE YELLOW BOOK

First list of clerical errors in Recommendation Z.200

CCITT High Level Language (CHILL)

1. Introduction

This paper is the first list of corrections of "clerical errors" in the CHILL definition, recommendation Z.200.

A "clerical error" is defined to be an error which, when corrected, does not change the interpretation that knowledgeable people would have given to the definition.

The number (1) in front of each correction refers to the fact that this is the first list of clerical errors. Possible future corrections will be included in this list and will be numbered (2), (3), etc.

2. List of corrections to clerical errors

(1) page 16, lines 10, 11

- replace :...if it is an M-value class or an M-derived class...

  by: ...if it is an M-value class or an M-derived class or an M-reference class...

(1) page 25, section 3.4.6

- add third static condition (which applies to the derived syntax for range modes) :

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

(1) page 27, section 3.6.4

- add a static condition :

  The referenced origin mode, if it is a structure mode, must be parameterizable

(1) page 42,

- in line 6,

  replace : With declarations and formal parameter.....

  by: With declarations, parameter and result .....

- replace syntax line (3)

  by : *<step> ::= (3)*

- replace syntax line (4)

  by : *<pos> ::= (4)*

(1) page 51, section 4.1.2

- replace first static condition

  by : 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*.

(1) page 55, section 4.2.2

- replace first dynamic condition

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

- replace in second dynamic condition : When accessing via *based name*...

  by : When accessing via a *based name*...

- in third dynamic condition, second line, underline: variant

(1) page 74, semantics, 3rd paragraph, last line

- replace: ....see section 9.1.4).

  by : ....see section 9.1.3).

(1) page 76, section 5.2.5, item 6, line 4

- replace : .... not *(ELSE)* must be...

  by : ...not *(ELSE)* nor *<irrelevant>* must be....

(1) page 82, section 5.2.10, static conditions, line 1:

- replace : .....must be strong The...

  by: .....must be strong. The ....

(1) page 87, section 5.2.16, semantics of *GETSTACK*, line 2

- replace : ....section 7.4..

  by: ....section 7.9...

- second static property, line 1

  replace : .... routine call is the class...

  by: .... routine call is the resulting class...

(1) page 88

- add a new static condition before : The array expression as an....:

  The *static mode location* argument of *SIZE*
  must be referable.

- the sixth static condition (about getstack argument), second part,
  should start as follows :

- The variant structure mode must be parameterizable and there must
  be as many expressions....

(1) page 105, line 4

- replace :... Each case label defines...

  by: ... Each case label list defines...

(1) page 107, syntax lines (6.1) and (8.1)

- replace : <*expression*>

  by: <*discrete expression*>

(1) page 108, line 12

- replace : do action.

  by: do action, or if the handler of the do action is entered
  and fall through, or if the do action is left by a return
  or a stop action.

(1) page 113, section 6.7, semantics, line 1

- replace : A call action causes...

  by: A call action causes either the call of a procedure or of
  a built-in routine. A procedure call causes...

(1) page 114, static conditions, fourth compatibility requirement (LOC attr.)

- add : If the *procedure call* is not regional,
  the (actual) *location* must not be regional
  (see section 8.2.2).

(1) page 123, section 6.19.2, semantics, line 10 from bottom

- replace :.....introduced value names....

  by: .....introduced value receive names...

(1) page 127, section 7.1, lines 1 and 2

- replace :....,*region, delay case action, receive case action*,..

  by: ...,*region, receive case action*,..

(1) page 136, section 7.4

- add the following static condition :

  All names mentioned in *exception list* must be different.

(1) page 137, section 7.7, semantics, line 2

- replace :....data object for....

  by: ....data objects for..

(1) page 141, section 8.2.1, line 8

- replace :....if and only if it...

  by: ....if either it...

(1) page 143, section 8.2.2, paragraph 2 (value), line 6

- replace : It is a *location contents* which is regional...

  by: It is a *location contents* of which the *location* contained in it is regional...

(1) page 145, section 8.4

- add new paragraph before "Receive buffer case action":

  Receive expression (see section 5.2.18)

  When a process evaluates a receive expression, it re-activates another process if and only if the set of delayed sending processes of the specified buffer location is not empty. In that case, it receives a value of the highest priority among the values in the buffer location or the delayed sending processes. Receiving a

value from a buffer, the process removes the value from the buffer and a delayed sending process with the value of the highest priority is selected to become active according to an implementation defined scheduling algorithm. This re-activated process is thus removed from the set of delayed sending processes and its value is stored in the buffer, with the specified priority. Receiving a value directly from a delayed sending process, the delayed process carrying the value with the highest priority is selected to become active according to an implementation defined scheduling algorithm. This re-activated process is thus removed from the set of delayed sending processes and its value is received.

(1) page 150, section 9.1.1.7, line 1

- replace :....is not a composite mode, has...

  by: ....is a discrete mode or a string mode, has...

(1) page 157, section 9.1.2.4, definition, line 1

- replace :.....is restrictable to a...

  by: .....is restrictable to a...

(1) page 158, section 9.1.2.5, rule 3, item 3, paragraph 3

- replace : if V is a variant structure mode....

  by: if V is a variant structure mode....

(1) page 159, section 9.1.2.6, rule 6, item 3, paragraph 3, last line

- replace :....denote the list of values of N.

  by: ....denote the list of values of M.

(1) page 180, section 11.6, line 1

- replace :....one syntatic description...

  by: ....one syntactic description...

(1) page 193, line numbered 34

- replace : *END stacks-1;*

by: *END stacks_1;*

(1) page 198

- insert between lines numbered 140 and 141 :

140a DCL c column;

(1) page 199

- replace lines numbered 28-32

by :
```
27a    MANIPULATE :
27b    MODULE
27c       SEIZE NODE, REMOVE, INSERT;
28        DCL NODE_A NODE :=(:NULL, NULL, 536 :);
29        REMOVE ( );
30        REMOVE ( );
31        INSERT (NODE_A);
31a    END MANIPULATE;
32 END CIRCULAR_LIST;
```

(1) page 201, line numbered 6

- replace :....lets calla through..

by: ....lets calls through...

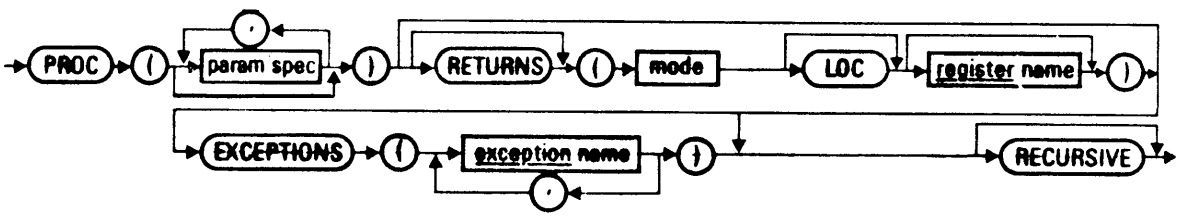
(1) page 202

- replace line numbered 15

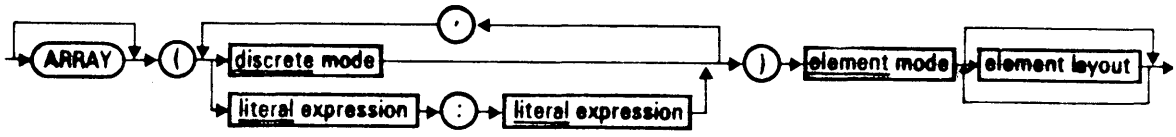
by : *15 ACQUIRE, RELEASE, CONGESTED, STEP, READOUT, READY;*

(1) page 214

- replace syntax diagram of PROC by:

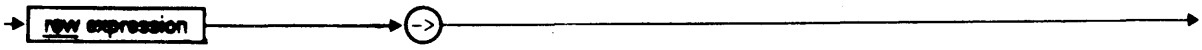


- replace syntax diagram of ARRAY by:



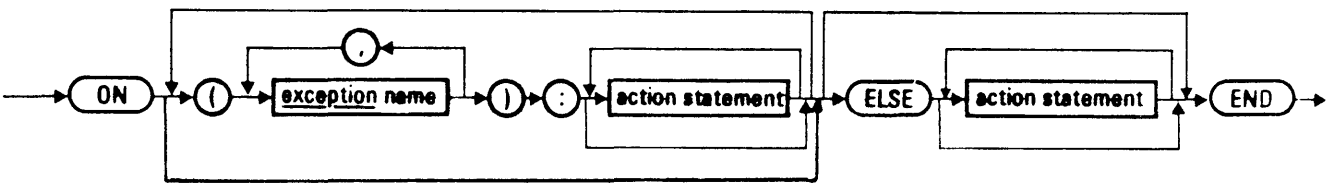
(1) page 216

- complete in the syntax diagram of location the box around row expression and the arrow leaving from it :



(1) page 219,

- replace syntax diagram of handler by:



(1) page 239

- replace : synonymouth with

  by: synonymous with

Printed in Switzerland — ISBN 92-61-01121-7

INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

BIBLIOTHÈQUE DE L' UIT GENÈVE

YELLOW BOOK

VOLUME VI - FASCICLE VI.8

CCITT HIGH LEVEL LANGUAGE (CHILL)

RECOMMENDATION Z.200

VIITH PLENARY ASSEMBLY
GENEVA, 10–21 NOVEMBER 1980

Geneva 1981

ISBN 92-61-01121-7

© I.T.U.

CONTENTS OF THE CCITT BOOK
APPLICABLE AFTER THE SEVENTH PLENARY ASSEMBLY (1980)

YELLOW BOOK

Volume I – Minutes and reports of the Plenary Assembly.

Opinions and Resolutions.

Recommendations on:
- the organization and working procedures of the CCITT (Series A);
- means of expression (Series B);
- general telecommunication statistics (Series C).

List of Study Groups and Questions under study.

Volume II

FASCICLE II.1 – General tariff principles – Charging and accounting in international telecommunications services. Serie D Recommendations (Study Group III).

FASCICLE II.2 – International telephone service – Operation. Recommendation E.100 - E.323 (Study Group II).

FASCICLE II.3 – International telephone service – Network management – Traffic engineering. Recommendations E.401 - E.543 (Study Group II).

FASCICLE II.4 – Telegraph and "telematic services"[1] operations and tariffs. Series F Recommendations (Study Group I).

Volume III

FASCICLE III.1 – General characteristics of international telephone connections and circuits. Recommendations G.101 - G.171 (Study Group XV, XVI, CMBD).

FASCICLE III.2 – International analogue carrier systems. Transmission media – characteristics. Recommendations G.211 - G.651 (Study Group XV, CMBD).

FASCICLE III.3 – Digital networks – transmission systems and multiplexing equipments. Recommendations G.701 - G.941 (Study Group XVIII).

FASCICLE III.4 – Line transmission of non telephone signals. Transmission of sound programme and television signals. Series H, J Recommendations (Study Group XV).

Volume IV

FASCICLE IV.1 – Maintenance; general principles, international carrier systems, international telephone circuits. Recommendations M.10 - M.761 (Study Group IV).

FASCICLE IV.2 – Maintenance; international voice frequency telegraphy and facsimile, international leased circuits. Recommendations M.800 - M.1235 (Study Group IV).

FASCICLE IV.3 – Maintenance; international sound programme and television transmission circuits. Series N Recommendations (Study Group IV).

FASCICLE IV.4 – Specifications of measuring equipment. Series O Recommendations (Study Group IV).

---

1) "Telematic services" is used provisionally.

Volume V – Telephone transmission quality. Series P Recommendations (Study Group XII).

Volume VI

FASCICLE VI.1 – General Recommendations on telephone switching and signalling. Interface with the maritime service. Recommendations Q.1 - Q.118 *bis* (Study Group XI).

FASCICLE VI.2 – Specifications of signalling systems Nos. 4 and 5. Recommendations Q.120 - Q.180 (Study Group XI).

FASCICLE VI.3 – Specifications of signalling system No. 6. Recommendations Q.251 - Q.300 (Study Group XI).

FASCICLE VI.4 – Specifications of signalling systems R1 and R2. Recommendations Q.310 - Q.480 (Study Group XI).

FASCICLE VI.5 – Digital transit exchanges for national and international applications. Interworking of signalling systems. Recommendations Q.501 - Q.685 (Study Group XI).

FASCICLE VI.6 – Specifications of signalling system No. 7. Recommendations Q.701 - Q.741 (Study Group XI).

FASCICLE VI.7 – Functional Specification and Description Language (SDL). Man-machine language (MML). Recommendations Z.101 - Z.104 and Z.311 - Z.341 (Study Group XI).

FASCICLE VI.8 – CCITT high level language (CHILL). Recommendation Z.200 (Study Group XI).

Volume VII

FASCICLE VII.1 – Telegraph transmission and switching. Series R, U Recommendations (Study Group IX).

FASCICLE VII.2 – Telegraph and "telematic services"[1] terminal equipment. Series S, T Recommendations (Study Group VIII).

Volume VIII

FASCICLE VIII.1 – Data communication over the telephone network. Series V Recommendations (Study Group XVII).

FASCICLE VIII.2 – Data communication networks; services and facilities, terminal equipment and interfaces. Recommendations X.1 - X.29 (Study Group VII).

FASCICLE VIII.3 – Data communication networks; transmission, signalling and switching, network aspects, maintenance, administrative arrangements. Recommendations X.40 - X.180 (Study Group VII).

Volume IX – Protection against interference. Series K Recommendations (Study Group V). Protection of cable sheaths and poles. Series L Recommendations (Study Group VI).

Volume X

FASCICLE X.1 – Terms and definitions.

FASCICLE X.2 – Index of the Yellow Book.

---

1) "Telematic services" is used provisionally.

CCITT HIGH LEVEL LANGUAGE (CHILL)

(GENEVA, 1980)

CONTENTS

1.0 INTRODUCTION

This recommendation defines the CCITT high level programming language CHILL. CHILL stands for CCITT High Level Language.

An alternative definition of CHILL, in a strict mathematical form, will be contained in CCITT Manual. Another CCITT Manual known as 'Introduction to CHILL' serves as an introduction to the language.

1.1 GENERAL

CHILL was designed primarily for programming SPC telephone exchanges. However, it is considered to be general enough for other applications (e.g. message switching, packet switching, modelling, etc.).

CHILL was designed with the following requirements in mind (refer to Question 8/XI of the study period 1977-1980):

- enhance reliability by allowing for extensive compile-time checking;
- permit the generation of highly efficient object code;
- be flexible and powerful in order to cover the required range of applications and to exploit various kinds of hardware;
- encourage modular and structured program development;
- be easy to learn and use.

CHILL does imply the existence of an environment for program development. This environment may implement, amongst other items, separate compilation, input/output and debugging tools. These items are not defined by this recommendation.

CHILL programs can be written in a machine independent manner for the class of machines known to be used, or proposed for use, in SPC telephone exchanges.

CHILL does not attempt to provide specific constructs for every application mentioned above, but rather it has a general base with a number of possibilities suitable for the particular application.

CHILL as a language is machine independent. A particular implementation may, however, contain implementation defined language objects. Programs containing such objects will in general, not be portable.

CHILL is designed under the assumption that it will be compiled from source text to object code. It is not specifically designed to make one-pass compilation feasible nor to minimise compiler size.

To allow security without an unacceptable loss of efficiency, much checking can be done statically. A few language rules can be tested only at run time. A violation of such a rule results in a run-time exception. However, the generation of run-time checks for these exceptions is optional, unless a programmer defined exception handler is specified.

1.2 LANGUAGE SURVEY

A CHILL program consists essentially of three parts:

- a description of data objects;
- a description of actions which are to be performed upon the data objects;
- a description of the program structure.

Data objects are described by data statements (declaration and definition statements), actions are described by action statements and the program structure is determined by program structuring statements.

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

CHILL provides for extensive static checking upon the usage of data objects in a given context.

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

1.3 MODES AND CLASSES

The manipulatable data objects of CHILL are values and locations where values can be stored.

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 the location and the values it may contain (note that not all properties of a location are determinable by its mode alone). Properties of locations are: size, internal structure, read-onlyness, 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 (symbolic) 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;
synchronisation modes	event and buffer modes for process synchronisation and communication.

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 only be determined dynamically. Dynamic modes are always parameterised modes with run-time parameters. A mode which is not dynamic, is called a static mode. An explicitly denoted mode in a CHILL program is always static.

Neither dynamic modes nor classes have a denotation in CHILL. They are only introduced in the metalanguage to describe static and dynamic context conditions.

1.4 LOCATIONS AND THEIR ACCESSES

Locations are (abstract) places where values can be stored or from which values can be obtained. In order to store or obtain a value, the 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;
3. based declarations.

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

Apart from location declarations, new locations can be created by means of a *GETSTACK* built-in routine which will yield a reference value (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 always 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 be read-only, which means that it can only be accessed to obtain a value and not to store a new value into it (except when initialising).

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 substringing, 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. (Note that the word dynamic is only used in relation to the mode; the location is not dynamic in the sense that it varies at run time; only that its properties cannot be completely determined statically.)

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.

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. Both a literal and a constant value are assumed to be evaluated before run time and cannot generate a run-time exception. A value may be regional, in which case it can refer somehow to regional locations. A value may be composite, i.e. containing 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 is not written in CHILL and with a more general parameter and result mechanism). To return from and/or establish the result of a procedure call, the result and return 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, similar to a decision table;

do action — for iteration or bracketing;

exit action — for leaving a bracketed action 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 or the evaluation of a receive expression.

1.7 PROGRAM STRUCTURE

The program structuring statements are the begin-end block, module, procedure, process and region. 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* 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 unauthorised 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 parameterised) sub-program which 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 and regions provide the means by which a structure of concurrent executions can be achieved.

A complete CHILL program is a list of modules or regions, which 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.

1.8 CONCURRENT EXECUTION

CHILL allows for the concurrent execution of program units. A process is the unit of concurrent execution. The start action causes the creation of a new process of the indicated process definition. The process is then

considered to be executed concurrently with the starting process. 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, causes its termination.

A process 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 process, the transition from delayed to active is called the re-activation of the process. The execution of delaying actions on events, or receiving actions on buffers or signals, or sending actions on buffers, can cause the executing process to become delayed. The execution of a continue action on events, or sending actions on buffers or signals, or receiving actions on buffers, can cause a delayed process to become active again.

Buffers and events are locations with restricted usage. 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 synchronising and transmitting information between processes. Events are only used for synchronisation. 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 synchronisation.

A region is a special kind of module. Its use is to provide for mutually exclusive access to data structures, which are shared by several processes.

1.9 *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 checking rules determine how names may be used, the scope checking rules determine where names may be used.

The mode checking 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 define the visibility of names which is determined by the program structure and explicit visibility statements. The explicit visibility statements determine the scope of the mentioned names and also of possibly implied names 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.

1.10 EXCEPTION HANDLING

The dynamic semantic conditions of CHILL are those (non context-free) conditions which, 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.) The violation of a dynamic semantic rule causes a run-time exception.

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 specifiable (i.e. it has a name) and 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.

Most exceptions have a name. This name 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 a CHILL defined exception name, the associated dynamic condition must be checked.

1.11 IMPLEMENTATION OPTIONS

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

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 which 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 is not specified in CHILL with a more general parameter passing and result transmission scheme than CHILL procedures.

A built-in process name is a process name whose definition is not specified in CHILL. A CHILL process may cooperate with implementation defined processes or start such processes.

An implementation defined exception handler is a handler appended to the imaginary outermost process definition. If this handler receives control after the occurrence of an exception, the implementation may decide which actions are to be taken.

2.0 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 italic 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 lefthand side of the symbol *::=*, and one or more constructs, consisting of non-terminal and/or terminal productions at the righthand side. These constructs are separated by a vertical bar (|) and 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 sub-category (see section 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 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 document and is not made to suit any particular parsing algorithm (e.g. there are some context-free ambiguities introduced in the interest of clarity).

2.1.2 THE SEMANTIC DESCRIPTION

For each syntactic category (non-terminal symbol), the semantic description is given in the 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 appropriate sections where the syntactic category is used.

When appropriate, a 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 section 2.1.1). This use requires the non-terminal to be of a specific semantic sub-category. E.g. ⟨*boolean expression*⟩ is identical to ⟨*expression*⟩ in the context free sense, but semantically it requires the *expression* to be of the boolean class. The underlined part is sometimes used in the text as an adjective to qualify the non-terminal. E.g. the sentence "the *expression* is constant" is identical to saying "the *expression* is a *constant expression*".

The section dynamic conditions describes the context-dependent conditions which must be fulfilled during execution. In some cases, conditions are static if and only if no dynamic modes are involved. In those cases, the condition is mentioned under static conditions and referred to under dynamic conditions.

In the semantic description the non-terminals are written in italics without the angular brackets to indicate the syntactic objects.

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 D. References indicate via which syntax rule each example is produced and from which example it is taken.

E.g. *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 names (see Appendix C). These special names 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 CCITT alphabet no. 5, Recommendation V.3 (see Appendix A1). It is possible to represent any CHILL program using a minimum character set which is a subset of the CCITT alphabet no.5 basic code (see Appendix A2).

The lexical elements of CHILL are:

- special symbols
- names
- literals

The special symbols are listed in Appendix B.

Names are formed according to the following syntax:

syntax:

<name> ::= (1)

*<letter> { <letter> | <digit> |_}** (1.1)

The underline symbol (_) forms part of the name, i.e. the name *LIFE_TIME* is different from the name *LIFETIME*. In the case that an alphabet with lower case letters is available, they may be used within names. Lower case and upper case letters are different, e.g. *Status* and *status* are two different names.

The language has a number of special names with predetermined meanings, see Appendix C. Some of them are reserved i.e. they cannot be used for other purposes unless explicitly freed by the free directive.

In the case that an alphabet with both upper an lower case letters is used, the special names may either all be in upper case representation or all be in lower case representation. The reserved names are only reserved in the chosen representation (e.g. if the lower case fashion is chosen, *row* is reserved, *ROW* is not).

2.3 THE USE OF SPACES

Spaces may be used to delimit the lexical elements of a program. Lexical elements are terminated by the first character that cannot be part of the lexical element. For instance, *IFBTHEN* will be considered a *name* 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 (/*). Contiguous spaces have the same delimiting effect as a single space.

2.4 COMMENTS

syntax:

<comment> ::= (1)
/ <character string> */* (1.1)

<character string> ::= (2)
*{<character>}** (2.1)

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

static properties: A *comment* may be inserted at all places where spaces are allowed as delimiters.

static conditions: The *character string* must not contain the special sequence: asterisk solidus (*/).

examples:

4.1 */* from collected algorithms from CACM nr.93 */* (1.1)

2.5 FORMAT EFFECTORS

The format effectors BS (Backspace), CR (Carriage return), FF (Form feed), HT (Horizontal tabulation), LF (Line feed), and VT (Vertical tabulation) of the CCITT alphabet no.5 (positions FE_0 to FE_5) are not mentioned in the CHILL context-free syntax description. However, an implementation may use these format effectors in CHILL programs. When used, they have the same delimiting effect as a space. They may not be used within lexical elements.

2.6 COMPILER DIRECTIVES

syntax:

```
<directive clause> ::=                                        (1)
        <> <directive> {,<directive>}* [<>]                   (1.1)

<directive> ::=                                               (2)
        <CHILL directive>                                     (2.1)
      | <implementation directive>                            (2.2)

<CHILL directive> ::=                                         (3)
        <free directive>                                      (3.1)

<free directive> ::=                                          (4)
        FREE(<reserved name list>)                            (4.1)

<name list> ::=                                               (5)
        <name> {,<name>}*                                     (5.1)
```

semantics: A directive clause conveys information to the compiler. Except for the free directive, 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 free directive applies to a compilation unit. It will free the reserved names specified in the *reserved name list* so that they may be redefined in the compilation unit.

static properties: A *directive clause* may be inserted at all places where spaces are allowed. 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 section 9.2.8).

static conditions: The optional directive-ending symbol (<>) may only be omitted if it is placed just in front of a semicolon (i.e. the *directive clause* is terminated with the first <> or semicolon. However, the semicolon does not belong to the *directive clause*. As a consequence, a *directive* may neither contain the symbol <> nor a semicolon unless placed between parentheses, see below). If parentheses occur in an *implementation directive*, they must be properly balanced and if a semicolon or the directive-ending symbol appears within parentheses, they do not end the *directive*.

examples:

```
15.1    <> FREE (STEP)                                        (1.1)
15.1    FREE (STEP)                                           (4.1)
```

3.0 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 which 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. This mode is always compatible with the class of the value and the value is one of the values defined by the mode. 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 parameterised modes with run-time parameters.

Static modes are denoted in the program by means of terminal productions of the syntactic category *mode*.

Dynamic modes have no denotations in CHILL. However, for description purposes, virtual denotations are introduced in this document to denote dynamic modes. These virtual denotations will be preceded by the ampersand symbol (&), i.e. *&VM(i)* denotes a parameterised dynamic mode with run-time parameter *i*.

In addition, in some places virtual denotations for static modes are introduced. This is done for modes which are not or cannot be explicitly denoted in the program text, but are virtualy introduced by some language constructs. These modes are also denoted by virtual denotations preceded by an ampersand.

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 any 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 any mode M with novelty nil (see section 9.1.1.1), 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 or an M-derived class, where M is a dynamic mode.

3.1.3 PROPERTIES OF, AND RELATIONS BETWEEN, MODES AND CLASSES

All fundamental properties of and relations between modes and classes are defined in chapter 9. The following gives a summary of these properties and relations:

1. A mode M has a novelty.

2. A mode M can be read-only.

3. A mode M can have the read-only property.

4. A mode M can have the referencing property.

5. A mode M can have the tagged parameterised property.

6. A mode M can have the synchronisation property.

7. A mode M can be defined by a mode N.

8. A mode M can be read-compatible with a mode N (asymmetric).

9. A mode M can be compatible with a class C (in that case C is said to be compatible with M).

10. A class C can have a root mode.

11. A class C can be compatible with a class D (symmetric).

12. Given a list of compatible classes, there exists the resulting class.

Specific properties are defined for each mode in the appropriate section. A property is said to be hereditary if, when it holds for a specific mode, it also holds for all mode names defined by that mode. Therefore,

hereditary properties will not be explicitly defined for mode names. Every property, which holds for a mode, also holds for that mode preceded by the keyword *READ* (except in some cases where the read-only property is involved: these cases are explicitly indicated). Therefore, properties will not be explicitly defined on modes preceded with *READ*.

3.2 MODE DEFINITIONS

3.2.1 GENERAL

syntax:

<mode definition> ::= (1)
<name list> = <defining mode> (1.1)

<defining mode> ::= (2)
<mode> (2.1)

derived syntax: A *mode definition* where the *name list* consists of more than one *name*, is derived from several mode definitions, one for each name, separated by comma's, with the same *defining mode*.

E.g. *NEWMODE DOLLAR, POUND = INT;* is derived from *NEWMODE DOLLAR = INT, POUND = INT;*

semantics: Mode definitions define one or more names to be a mode name, i.e. names denoting modes. Mode definitions occur inside newmode and synmode definition statements. The difference between a newmode and a synmode lies in the treatment by the mode equivalence algorithms (see section 9.1). All hereditary properties of the defining mode are, by definition, transferred to the defined mode name. Mode definitions may be (mutually) recursive.

static properties: A mode name is either one of the language defined mode names *INT, BOOL, CHAR, PTR, INSTANCE, EVENT,* or a name defined in a *mode definition*.

A mode name which is not one of the language defined mode names, has a unique defining mode, which is the mode denoted by the *defining mode* in the *mode definition* in which it is defined.

A set of recursive definitions is a set of mode definitions or synonym definitions (see section 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 or a set element 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 set of recursive definitions must be a set of recursive mode definitions; see section 5.1).

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: *NEWMODE M = STRUCT(i M, n 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* or a *row mode*, or a *procedure mode* at the marked place. The mode must be, or be contained in, the *defining mode* of the predecessor of the *mode name*.

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:

```
1.12    operand_mode = INT                                  (1.1)
3.3     complex = STRUCT(re,im INT)                         (1.1)
```

3.2.2 SYNMODE DEFINITIONS

syntax:

```
<synmode definition statement> ::=                          (1)
     SYNMODE <mode definition> {, <mode definition>}*;      (1.1)
```

semantics: Synmode definition statements define names to denote modes which are synonymous with their defining mode. The precise treatment of names defined in a synmode definition is explained in section 9.1.

static properties: A *name* is said to be a synmode name, if and only if it is defined in a *mode definition* in a *synmode definition statement*. A synmode name is said to be synonymous with a given mode, (conversely, the given mode is said to be

synonymous with the synmode name) if and only if:

- either the given mode is the *defining mode* of the synmode name;
- or the *defining mode* of the synmode name is itself a synmode name, synonymous with the given 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:

```
<newmode definition statement> ::=                           (1)
    NEWMODE <mode definition> {, <mode definition>}*;        (1.1)
```

semantics: Newmode definition statements define names to denote modes which are not synonymous with the defining mode. The values defined by a newmode are the values defined by the defining mode. The precise treatment of names defined in a newmode definition statement is explained in section 9.1.

static properties: A *name* is said to be a newmode name if and only if it is defined in a *mode definition* in a *newmode definition statement*.

If the *defining mode* is a range mode, then, together with the defined newmode name, a new virtual name is introduced, denoted by *&name_parent*, denoting the parent mode of the newmode name. The values defined by this virtual parent mode are the values of the parent mode of the defining range mode. The upper bound and lower bound of the virtual parent mode are the ones of the parent mode of the defining *range mode*.

If the *defining mode* is a string mode, then the new virtual modes: *&name(i)* are introduced for each *i* larger than the string length of the newmode name, where *&name* denotes the introduced newmode. This *i* denotes the string length of the virtual mode. The hereditary property bit or character string of the newmode name is transferred to the virtual modes.

examples:

```
11.4    NEWMODE line = INT(1:8);                             (1.1)
11.10   NEWMODE board = ARRAY(line) ARRAY(column) square;    (1.1)
```

3.3 MODE CLASSIFICATION

syntax:

```
<mode> ::=                                          (1)
      <non-composite mode>                          (1.1)
    | <composite mode>                              (1.2)

<non-composite mode> ::=                            (2)
      <discrete mode>                               (2.1)
    | <powerset mode>                               (2.2)
    | <reference mode>                              (2.3)
    | <procedure mode>                              (2.4)
    | <instance mode>                               (2.5)
    | <synchronisation mode>                        (2.6)
```

semantics: Modes are denoted in a CHILL program by the terminal productions of the syntactic category *mode*. In the sequel of this chapter, the specific properties of the different modes will be defined. The equality (=) and inequality (/=) relations are defined on the set of values of any given mode (see section 5.3).

static properties: A *mode* has a size, which is the value delivered by *SIZE(M)*, where *M* is a virtual synmode name synonymous with *mode*.

3.4 DISCRETE MODES

3.4.1 GENERAL

syntax:

```
<discrete mode> ::=                                 (1)
      <integer mode>                                (1.1)
    | <boolean mode>                                (1.2)
    | <character mode>                              (1.3)
    | <set mode>                                    (1.4)
    | <range mode>                                  (1.5)
```

semantics: Discrete modes define sets and subsets of well-ordered values. All discrete modes, which are not range modes, can be parent modes of range modes (see section 3.4.6). All discrete modes define an upper bound and a lower bound, denoting the highest and lowest value, respectively.

3.4.2 *INTEGER MODES*

syntax:

```
<integer mode> ::=                                   (1)
     [READ] INT                                      (1.1)
   | [READ] BIN                                      (1.2)
   | [READ] <integer mode name>                      (1.3)
```

derived syntax: *BIN* is derived syntax for *INT*.

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 section 5.3.2). An implementation may define other integer modes with different bounds (e.g. *LONG_INT, SHORT_INT, ...*) which may also be used as parent modes for ranges (see section 11.2).

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

- The upper bound and lower bound of an integer mode are the literals denoting respectively the highest and lowest value defined by the integer mode.
- The number of values of an integer mode is implementation defined.

examples:

```
1.4      INT                                         (1.1)
```

3.4.3 *BOOLEAN MODES*

syntax:

```
<boolean mode> ::=                                   (1)
     [READ] BOOL                                     (1.1)
   | [READ] <boolean mode name>                      (1.2)
```

semantics: A boolean mode defines the logical truth values (*TRUE* and *FALSE*), with the usual boolean operations (see section 5.3.2). *TRUE* is greater than *FALSE*.

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

- The upper bound of a boolean mode is *TRUE*, its lower bound is *FALSE* .
- The number of values defined by a boolean mode is 2.

examples:

5.4 BOOL (1.1)

3.4.4 CHARACTER MODES

syntax:

<character mode> ::= (1)
[READ] CHAR (1.1)
| [READ] <character mode name> (1.2)

semantics: A character mode defines the character values as described by the CCITT alphabet no.5, International reference version (Recommendation V3, see Appendix A1). This alphabet also defines the ordering of the characters.

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

- The upper bound and lower bound of a character mode are the character string literals of length 1 denoting respectively the highest and lowest value defined by CHAR.
- The number of values defined by a character mode is 128.

examples:

8.4 CHAR (1.1)

3.4.5 SET MODES

syntax:

<set mode> ::= (1)
[READ] SET(<set list>) (1.1)
| [READ] <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)
<name> = <integer literal expression> (4.1)

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

```
<set element> ::=                                   (6)
     <name>                                         (6.1)
   | <unnamed value>                                (6.2)

<unnamed value> ::=                                 (7)
     *                                              (7.1)
```

semantics: A set mode defines a set of named or unnamed values. The named values are denoted by the names in the *set list*; the unnamed values are the other values. The internal representation of the named values is the integer value associated with the named value (see below). This representation also defines the ordering of the values.

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

- A set mode has a set of set element names which is the set of element names in its *set list*.
- Each set element name of a set mode has an integer (representation) value attached which is, in the case of a *numbered set list*, the value delivered by the *integer literal expression* in the *numbered set element* in which the set element name occurs, otherwise one of the values 0,1,2,.... etc., according to its position in the *unnumbered set list*. For example: *SET(*,A,*,B,*)*, *A* has representation value 1 and *B* representation value 3 attached.
- A set mode has an upper bound and a lower bound which are its set element names which denote the highest and lowest named values, respectively.
- The number of values of a set mode is, in the case of a *numbered set list*, the highest of the values attached to the set element names plus 1, otherwise the number of *set element* occurrences in the *unnumbered set list*.
- A set mode is a set mode with holes, if and only if the number of *name* occurrences in the *set list* is less than the number of values of the set mode.

static conditions: Each *integer literal expression* in the *set list* must deliver a different non-negative integer value in the sense that for any two expressions *e1* and *e2*: *NUM(e1)* and *NUM(e2)* deliver different results.

A set mode must define at least one named value.

examples:

```
11.5    SET( occupied, free)                        (1.1)
6.4     month                                       (1.2)
```

3.4.6 *RANGE MODES*

syntax:

```
<range mode> ::=                                                    (1)
     [READ] <discrete mode name>( <literal range> )                 (1.1)
   | [READ] RANGE( <literal range> )                                (1.2)
   | [READ] BIN( <integer literal expression> )                     (1.3)
   | [READ] <range mode name>                                       (1.4)

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

<lower bound> ::=                                                   (3)
     <discrete literal expression>                                  (3.1)

<upper bound> ::=                                                   (4)
     <discrete literal expression>                                  (4.1)
```

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

semantics: A 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, which determines the operations on and ordering of the range values.

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

- If the range mode is of the form:
  <discrete mode name>(<literal range>)
  then if the *discrete mode name* is not a range mode then the parent mode is the *discrete mode name*, otherwise it is the parent mode of the *discrete mode name*.

- If the range mode is of the form:
  RANGE(<literal range>)
  then the parent mode is the root mode of the resulting class of the classes of the *upper bound* and *lower bound* in the *literal range*.

- If the range mode is a *synmode name*, then its parent mode is that of the defining mode of the *synmode name*.

- If the range mode is a *newmode name*, then its parent mode is the virtually introduced parent mode (see section 3.2.3).

A range mode has the following hereditary properties:

- A range mode has a lower bound and an upper bound which are the literals denoting the values delivered by *lower bound* and *upper bound* respectively in the *literal range*.

- The number of values of a range mode is the value delivered by *NUM(U) - NUM(L) + 1*, where *U* and *L* denote respectively the upper bound and lower bound of the range mode.

- A range mode is said to be a range mode with holes, if and only if its parent mode is a set mode with holes and an unnamed value is in the range specified by the range 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 which is less than or equal to the value delivered by *upper bound*, and both values must lie in the value range defined by *discrete mode name*, if specified.

examples:

9.4	*INT(2:max)*	*(1.1)*
11.11	*line*	*(1.4)*
9.4	*2:max*	*(2.1)*

3.5 POWERSET MODES

syntax:

```
<powerset mode> ::=                                   (1)
     [READ] POWERSET <member mode>                    (1.1)
   | [READ] <powerset mode name>                      (1.2)

<member mode> ::=                                     (2)
     <discrete mode>                                  (2.1)
```

semantics: A powerset mode defines values which 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 section 5.3).

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

- It has a unique member mode which is the mode denoted by *member mode*.

examples:

8.4	*POWERSET CHAR*	*(1.1)*
9.4	*POWERSET INT(2:max)*	*(1.1)*
9.6	*number_list*	*(1.2)*

3.6 REFERENCE MODES

3.6.1 GENERAL

syntax:

```
<reference mode> ::=                                   (1)
      <bound reference mode>                           (1.1)
    | <free reference mode>                            (1.2)
    | <row mode>                                       (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; 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 sections 4.2.3, 4.2.4 and 4.2.15), delivering the location which 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*).

3.6.2 BOUND REFERENCE MODES

syntax:

```
<bound reference mode> ::=                             (1)
      [READ] REF <referenced mode>                     (1.1)
    | [READ] <bound reference mode name>               (1.2)

<referenced mode> ::=                                  (2)
      <mode>                                           (2.1)
```

semantics: Bound references define reference values to locations of the specified referenced mode.

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

- It has a unique referenced mode which is the mode denoted by *referenced mode*.

examples:

```
10.38   REF cell                                       (1.1)
```

3.6.3 *FREE REFERENCE MODE*

syntax:

<free reference mode> ::= (1)
[READ] PTR (1.1)
| *[READ] <free reference mode name>* (1.2)

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

examples:

19.5 PTR (1.1)

3.6.4 *ROW MODES*

syntax:

<row mode> ::= (1)
[READ] ROW <string mode> (1.1)
| *[READ] ROW <array mode>* (1.2)
| *[READ] ROW <variant structure mode name>* (1.3)
| *[READ] <row mode name>* (1.4)

semantics: A row mode defines reference values to locations of dynamic mode (which are locations of some parameterised mode with statically unknown parameters).

A row value may refer to:

string locations with statically unknown length,

array locations with statically unknown upper bound,

parameterised structure locations with statically unknown parameters.

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

- It has a referenced origin mode, which is the *string mode*, the *array mode*, or the *variant structure mode name*, respectively.

examples:

8.6 ROW CHAR (max) (1.1)

3.7 PROCEDURE MODES

syntax:

```
<procedure mode> ::=                                              (1)
     [READ] PROC( [<parameter list>] ) [<result spec>]
     [EXCEPTIONS( <exception list> )] [RECURSIVE]                 (1.1)
   | [READ] <procedure mode name>                                 (1.2)

<parameter list> ::=                                              (2)
     <parameter spec> {,<parameter spec>}*                        (2.1)

<parameter spec> ::=                                              (3)
     <mode> [<parameter attribute>] [<register name>]             (3.1)

<parameter attribute> ::=                                         (4)
     IN | OUT | INOUT | LOC                                       (4.1)

<result spec> ::=                                                 (5)
     [RETURNS] (<mode> [LOC] [<register name>])                   (5.1)

<exception list> ::=                                              (6)
     <exception name> {,<exception name>}*                        (6.1)

<exception name> ::=                                              (7)
     <name>                                                       (7.1)
```

derived syntax: A *result spec* without the optional keyword *RETURNS* is derived syntax for the *result spec* with *RETURNS*.

semantics: A procedure mode defines (general) procedure values, i.e. the objects denoted by general procedure names which are names defined in procedure definition statements or entry definition statements. The 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 section 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:

- It has a list of parameter specs, each parameter spec consisting of a mode, possibly a parameter attribute and/or register name. The parameter specs are defined by the *parameter list*.

- It has an optional result spec, consisting of a mode, an optional *LOC* attribute and/or register name. The result spec is defined by the *result spec*.

- It has a possibly empty set of exception names, which are the names mentioned in the *exception list*.

- It has a recursivity which is recursive if *RECURSIVE* is specified, otherwise an implementation defined default specifies either recursive or non-recursive.

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

Only if *LOC* is specified in the *parameter spec* or *result spec*, may the *mode* in it have the synchronisation property.

3.8 *INSTANCE MODES*

syntax:

```
<instance mode> ::=                                   (1)
    [READ] INSTANCE                                   (1.1)
  | [READ] <instance mode name>                       (1.2)
```

semantics: An instance mode defines values which uniquely identify processes. The creation of a new process (see section 5.2.17 and 8.1) 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:

```
15.29   INSTANCE                                      (1.1)
```

3.9 *SYNCHRONISATION MODES*

3.9.1 *GENERAL*

syntax:

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

semantics: Locations of synchronisation mode provide the means of synchronisation and communication between processes (see chapter 8). There exists no expression in CHILL denoting a value defined by a synchronisation mode. As a consequence, there are no operations defined on the values.

3.9.2 EVENT MODES

syntax:

```
<event mode> ::=                                          (1)
     [READ] EVENT [(<event length>)]                      (1.1)
   | [READ] <event mode name>                             (1.2)

<event length> ::=                                        (2)
     <integer literal expression>                         (2.1)
```

semantics: Event mode locations provide the means for synchronisation 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 section 6.15, 6.16 and 6.17 respectively.

static properties: An event mode has the following hereditary property:

- It has possibly an event length attached, which is the value delivered by *NUM(event length)*.

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

examples:

```
14.10   EVENT                                             (1.1)
```

3.9.3 BUFFER MODES

syntax:

```
<buffer mode> ::=                                         (1)
     [READ] BUFFER [(<buffer length>)]
     <buffer element mode>                                (1.1)
   | [READ]<buffer mode name>                             (1.2)

<buffer length> ::=                                       (2)
     <integer literal expression>                         (2.1)

<buffer element mode> ::=                                 (3)
     <mode>                                               (3.1)
```

N.B. The syntax given above is syntactically ambiguous in connection with the syntax of the array modes. The following default interpretation applies: if the keyword *BUFFER* is immediately followed by an opening parenthesis, the text immediately following it is considered to be the start of the optional *buffer length* indication and not as belonging to the *buffer element mode*.

semantics: Buffer mode locations provide the means of synchronisation and communication between processes. The operations defined on buffer locations are the send action, the receive case action and the receive expression, described in section 6.18, 6.19 and 5.2.18 respectively.

static properties: A buffer mode has the following hereditary properties attached:

- It has an optional buffer length, which is the value delivered by *NUM(buffer length)*.
- It has a buffer element mode, which is the mode denoted by *buffer element mode*.

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

The *buffer element mode* must not have the synchronisation property.

examples:

```
16.28   BUFFER(1) USER_MESSAGES                    (1.1)
16.32   USER_BUFFERS                               (1.2)
```

3.10 COMPOSITE MODES

3.10.1 GENERAL

syntax:

```
<composite mode> ::=                               (1)
      <string mode>                                (1.1)
    | <array mode>                                 (1.2)
    | <structure mode>                             (1.3)
```

semantics: Composite locations and values have sub-locations and sub-values which can be accessed or obtained respectively (see sections 4.2.5-9, 4.2.13-14 and 5.2.6-12).

syntax:

```
<string mode> ::=                                                    (1)
     [READ] <string type>( <string length> )                         (1.1)
   | <parameterised string mode>                                      (1.2)
   | [READ] <string mode name>                                        (1.3)

<parameterised string mode> ::=                                      (2)
     [READ] <origin string mode name>( <string length> )             (2.1)
   | [READ] <parameterised string mode name>                          (2.2)

<origin string mode name> ::=                                        (3)
     <string mode name>                                              (3.1)

<string type> ::=                                                    (4)
     CHAR                                                            (4.1)
   | BIT                                                             (4.2)

<string length> ::=                                                  (5)
     <integer literal expression>                                    (5.1)
```

semantics: A string mode defines bit or character string values of a length indicated or implied by the string mode.

The string values of a given string mode are well-ordered. For character string values the ordering is the lexicographical order as defined by the CCITT alphabet no. 5. For bit string values the ordering is the lexicographical order such that a bit which is 1, is greater than a bit which is 0.

The concatenation operator is defined on string values. The usual logical operators are defined on bit string values (see section 5.3).

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

- It is a bit string mode or a character string mode, depending on whether *string type* specifies *BIT* or *CHAR*, or whether *origin string mode name* is a bit or character string mode.

- It has a string length, which is the value delivered by *NUM(string length)*.

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

The value delivered by the *string length* directly contained in a *parameterised string mode* must be less than or equal to the string length of the *origin string mode name*.

examples:

7.45 CHAR (20) (1.1)

3.10.3 ARRAY MODES

syntax:

```
<array mode> ::=                                                    (1)
     [READ] [ARRAY] (<index mode> {,<index mode>}*)
     <element mode> {<element layout>}*                             (1.1)
   | <parameterised array mode>                                     (1.2)
   | [READ] <array mode name>                                       (1.3)

<parameterised array mode> ::=                                      (2)
     [READ] <origin array mode name>( <upper index> )               (2.1)
   | [READ] <parameterised array mode name>                         (2.2)

<origin array mode name> ::=                                        (3)
     <array mode name>                                              (3.1)

<index mode> ::=                                                    (4)
     <discrete mode>                                                (4.1)
   | <literal range>                                                (4.2)

<upper index> ::=                                                   (5)
   <literal expression>                                             (5.1)

<element mode> ::=                                                  (6)
   <mode>                                                           (6.1)
```

derived syntax: The keyword *ARRAY* is optional. An *array mode* (which is neither an *array mode name* nor a *parameterised array mode*) without the keyword *ARRAY*, is derived from the *array mode* with the keyword *ARRAY*.

The index mode notation *<literal range>* is derived from the discrete mode *RANGE(<literal range>)*. 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* which is an *array mode*. For example:
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 section 3.10.6). Two array values are equal if and only if all corresponding element values are equal.

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

- It has an index mode which is the discrete mode denoted by *index mode* if it is not a *parameterised array mode*, otherwise the index mode is the 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*.

- It has an upper bound and a lower bound which are respectively the upper bound and the lower bound of its index mode.

- It has 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.

- It has an element layout which, if it is a *parameterised 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*.

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

- It has a number of elements which is the value delivered by:
  NUM(upper bound) - NUM(lower bound) + 1

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.

The *index mode* must not be a set mode with holes nor a range mode with holes.

examples:

5.30	*ARRAY(1:16) STRUCT(c4, c2, c1 BOOL)*	*(1.1)*
11.10	*ARRAY(line) ARRAY(column) square*	*(1.1)*
11.15	*board*	*(1.3)*

3.10.4 STRUCTURE MODES

syntax:

```
<structure mode> ::=                                              (1)
      <nested structure mode>                                     (1.1)
    | <level structure mode>                                      (1.2)
    | <parameterised structure mode>                              (1.3)
    | [READ] <structure mode name>                                (1.4)

<nested structure mode> ::=                                       (2)
      [READ] STRUCT (<fields> {,<fields>}*)                       (2.1)

<fields> ::=                                                      (3)
      <fixed fields>                                              (3.1)
    | <alternative fields>                                        (3.2)

<fixed fields> ::=                                                (4)
      <name list> <mode> [<field layout>]                         (4.1)

<alternative fields> ::=                                          (5)
      CASE [<tags>] OF
      <variant alternative> {,<variant alternative>}*
      [ELSE [<variant fields> {,<variant fields>}*]] ESAC         (5.1)

<variant alternative> ::=                                         (6)
      [<case label specification>]
      : [<variant fields> {,<variant fields>}*]                   (6.1)

<tags> ::=                                                        (7)
      <tag field name> {,<tag field name>}*                       (7.1)

<variant fields> ::=                                              (8)
      <name list> <mode> [<field layout>]                         (8.1)

<parameterised structure mode> ::=                                (9)
      [READ] <origin variant structure mode name>
      (<literal expression list>)                                 (9.1)
    | [READ] <parameterised structure mode name>                  (9.2)

<origin variant structure mode name> ::=                          (10)
      <variant structure mode name>                               (10.1)

<literal expression list> ::=                                     (11)
      <literal expression> {,<literal expression>}*               (11.1)
```

derived syntax: A *level structure mode* is derived syntax for a *nested structure mode* . This is explained in section 3.10.5.

A *fixed fields* occurrence or *variant fields* occurrence, where *name list* consists of more than one *name*, is derived syntax for several *fixed fields* occurrences or *variant fields* occurrences with one *name* 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:
STRUCT(I,J BOOL PACK)
is derived from:
STRUCT(I 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 which 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 parameterised structures.

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

Variant structures have variant fields, i.e. fields which 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.

A parameterised 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 parameterised structure and may not change during run time. The tag fields, if present, are read-only and automatically initialised with the specified values. For a parameterised structure location, a precise amount of storage can be allocated at the point of declaration or generation. Note that also (virtual) dynamic parameterised structure modes exist. Their semantics are defined in section 3.11.4.

The layout of a structure location or value can be controlled by means of a field layout specification (see section 3.10.6).

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

static properties:

general:

A structure mode has the following hereditary properties:

- A structure mode is a fixed structure mode if and only if it is denoted by a *nested* (or *level*) *structure mode* which does not directly contain an *alternative fields* occurrence.

- A structure mode is a variant structure mode if and only if it is denoted by a *nested* (or *level*) *structure mode* and contains at least one *alternative fields* occurrence.

- A structure mode is a parameterised structure mode if and only if it is denoted by a *parameterised structure mode*.

- A structure mode has a set of field names. This set is determined below for the different cases. A name is said to be a field name if and only if it is defined in a *name list* in *fixed fields* or *variant fields* in a *structure mode*. Each field name of a given structure mode has a unique field mode attached to it, which is either *M* or *READ M*, where *M* is the *mode* following the field name. The field mode will be *READ M* if the *mode* following is not a read-only mode and either it is the tag field name of a parameterised structure mode (see below), or the *structure mode* is a read-only mode.

  A field name of a given structure mode has a unique field layout attached to it which is the *field layout* following the field name, if present, otherwise the default field layout, which is either *PACK* or *NOPACK*. A field name is (language) referable if and only if its field layout is *NOPACK*.

- A *structure mode* denotes a mapped mode if and only if its field names have a field layout which is *pos*.

fixed structures:

A fixed structure mode has the following hereditary property:

- It has a set of field names which is the set of names defined by any *name list* in *fixed fields*. These field names are fixed field names.

variant structures:

A variant structure mode has the following hereditary properties:

- It has a set of field names, which is the union of the set of names defined by any *name list* in *fixed fields* and the set of names defined by any *name list* in *alternative*

fields. Field names defined by a *name 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 *tags* of an *alternative fields*. *alternative fields* in which no *tags* are specified, are tag-less *alternative fields*. The variant field names defined by any *name list* in *variant fields* of a tag-less *alternative fields* are tag-less variant field names. The other variant field names are tagged variant field names.

- A variant structure mode is a tag-less variant structure mode if and only if all its *alternative fields* occurrences are tag-less. Otherwise it is a tagged variant structure mode.

- A variant structure mode is a parameterisable variant structure mode if and only if it is either a tagged variant structure mode or a tag-less variant structure mode where for each of the *alternative fields* occurrences a *case label specification* is given for all the *variant alternative* occurrences in it.

- A parameterisable variant structure mode has a list of classes attached, determined as follow:

  - 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 as 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 fields* by concatenating them in the order as the *alternative fields* occur. The resulting list of classes of an *alternative fields* occurrence is the resulting list of classes of the list of *case label specification* occurrences in it (see section 9.1.3).

parameterised structures:

A parameterised structure mode has the following hereditary properties:

- It has an origin variant structure mode, which is the mode denoted by *origin variant structure mode name*

- It is a tagged parameterised structure mode if and only if its origin variant structure mode is a tagged variant structure mode, otherwise the parameterised structure mode is tag-less.

- It has 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, which are defined in *variant alternative* occurrences which are selected by the list of values defined by *literal expression list*.

  The set of tag field names of a *parameterised structure mode* is the set of tag field names of its origin variant structure mode.

- A parameterised structure mode has a list of values attached, defined by *literal expression list*.

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 fields* occurrences in whose *tags* 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.

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

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 section 9.1.3), and the same completeness, consistency and compatibility requirements must hold as for the case action (see section 6.4). Each of the tag field names of *tags* (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 parameterisable 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.)

parameterised structures:

The *origin variant structure mode name* must be parameterisable.

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)                                  (2.1)
11.5    STRUCT(status SET(occupied, free),
             CASE status OF
              (occupied): p piece,
              (free):
              ESAC)                                         (2.1)
2.5     fraction                                            (1.4)
11.5    status SET(occupied, free)                          (4.1)
11.6    status                                              (7.1)
11.7    p piece                                             (8.1)
```

3.10.5 *LEVEL STRUCTURE NOTATION*

derived syntax:

```
<level structure mode> ::=                                  (1)
     1 [<array specification>]
     [READ] {,<(2) level fields>}*                          (1.1)

<(n) level fields> ::=                                      (2)
     <(n) level fixed fields>                               (2.1)
   | <(n) level alternative fields>                         (2.2)

<(n) level fixed fields> ::=                                (3)
     n <name list> <mode> [<field layout>]                  (3.1)
   | n <name list> [<array specification>]
     [READ] [<field layout>] {,(n+1) level fields>}*        (3.2)

<(n) level alternative fields> ::=                          (4)
     CASE [<tags>] OF
     <(n) level alternative> {,<(n) level alternative>}*
     [ELSE [<(n) level variant fields>
     {,<(n) level variant fields>}*]]
     ESAC                                                   (4.1)

<(n) level alternative> ::=                                 (5)
     [<case label specification>
     {,<case label specification>}*]
```

```
    : [<(n) level variant fields>
    {,<(n) level variant fields>}*]                          (5.1)

<(n) level variant fields> ::=                               (6)
    n <name list> <mode> [<field layout>]                    (6.1)
  | n <name list> [<array specification>]
    [READ] [<field layout>] {,<(n+1) level fields>}*         (6.2)

<array specification> ::=                                    (7)
    [READ] [ARRAY] (<index mode> {,<index mode>}*)
    {<element layout>}*                                      (7.1)
```

N.B. The above description of a level number notation for structures involves an extension to the syntax description method explained in chapter 2: the syntax is recursively defined using the structuring level number *(n)* as parameter.

semantics: The *level structure mode* is derived syntax for a unique *nested structure mode*.

The nested notation is considered as strict syntax and all semantics, properties and conditions are explained in terms of it (see section 3.10.4).

If a structure contains fields which are themselves structures or arrays of structures, a hierarchy of structures is formed and a level number can be associated with each field.

Example:

```
SYNMODE M = STRUCT(B BOOL,
                   S ARRAY (1:10) STRUCT (T INT, U BOOL));
```

The structure as a whole has level 1, *B* and *S* have level 2, *T* and *U* have level 3. Instead of writing nested structure modes, it is allowed in the *level structure mode* to write the level number in the front of the name.

Example:

```
SYNMODE M = 1, 2 B BOOL,
               2 S ARRAY (1:10),
                 3 T INT,
                 3 U BOOL;
```

In mode definitions and synonym definitions with a mode there is no name associated with the first level. The association occurs at the declaration or at the point of formal parameter specification. At these places, the name of the first level will be placed after the level-1 position.

Example:

```
DCL 1 A,
      2 B BOOL,
      2 S ARRAY (1:10),
        3 T INT,
        3 U BOOL;
```

With declarations and formal parameter specifications, attributes and initialisations, if present, must be specified at the end of the level-1 position.

Example:

```
P : PROC (1 X INOUT,
            2 B BOOL,
            2 C INT);
```

If within a *level structure mode* an array of structures is specified, the array specification is given behind after the level indicator.

static conditions: Nested and level notations must not be mixed.

examples:

```
19.9    DCL 1   BASED (P),
              2 I INFO POS(0,8:31),
              2 PREV PTR POS(1,0:15),
              2 NEXT PTR POS(1,16:31)                    (1.1)
```

3.10.6 LAYOUT DESCRIPTION FOR ARRAY MODES AND STRUCTURE MODES

syntax:

```
<element layout> ::=                                     (1)
     PACK | NOPACK | <step>                              (1.1)

<field layout> ::=                                       (2)
     PACK | NOPACK | <pos>                               (2.1)

<step>                                                   (3)
     STEP(<pos> [,<step size> [,<pattern size>]])        (3.1)

<pos>                                                    (4)
     POS(<word> ,<start bit> ,<length>)                  (4.1)
   | POS(<word> [,<start bit> [: <end bit>]])            (4.2)

<pattern size> ::=                                       (5)
     <integer literal expression>                        (5.1)

<word> ::=                                               (6)
     <integer literal expression>                        (6.1)

<step size> ::=                                          (7)
```

<integer literal expression> (7.1)

<start bit> ::= (8)
<integer literal expression> (8.1)

<end bit> ::= (9)
<integer literal expression> (9.1)

<length> ::= (10)
<integer literal expression> (10.1)

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 in either *PACK* or *NOPACK*, mapping information is either a *step* in the case of array modes, or *pos*, in the case of fields of structure modes. The absence of *element layout* or *field layout* in an array or structure mode will always be interpreted as packing information, i.e. either as *PACK* or as *NOPACK*.

If *PACK* is specified for elements of an array or field of a structure, it means that the use of memory space is optimised for the array elements or structure fields, whereas *NOPACK* implies that the access time for the array elements or the structure fields is optimised. *NOPACK* also implies (language) referability.

The *PACK*, *NOPACK* information is only applied 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 *NOPACK*:
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 a composite object 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.

The precise positioning of C, a component i.e. element or field of an object, is given by the following three constants: W_c, B_c and L_c where

W_c is the distance in words of the first word which is (maybe partially) occupied by C, relative to the first word which is (maybe partially) occupied by the object of which is C a component,

B_c is the distance in bits of the first bit which is occupied by C, relative to the leftmost bit of the first word which is (maybe partially) occupied by C,

L_c is the number of bits which are occupied by C.

The positioning information, given for the components of the object, determines the precise positioning of these components, if the object is entire (i.e. it is not a component of another object). However, if the object is not entire, then the precise positioning of the component is dependent on the precise positioning of the object itself.

A *step* specified for the elements of an array is a shorthand notation for the explicit enumeration of the *pos* of each individual element. Informally, the *pos* and the *step size* specify a "positioning pattern" for the elements which completely fit in the first *pattern size* words, assuming the array to be entire. The positioning of the first element is determined by *pos*; the positioning of the subsequent elements which fit completely in the first *pattern size* words, is such that the distance in bits between the first occupied bits of successive elements is *step size*. The positioning pattern specified this way is repeated as often as needed for subsequent units of *pattern size* words.

Pos

Given an object O of a mapped mode in which a *pos* of the form:
POS(<word number>, <start bit> , <length>)
is specified for a component C of that object, the precise positioning of the the component C is determined as follows:

- If the object O (of which C is a component) is entire, then

  W_c is *NUM(word number)*

  B_c is *NUM(start bit)*, and

  L_c is *NUM(length)*.

- If the object O (of which C is a component) is not entire, then

  W_c is:
  NUM(word number) + (NUM(start bit) + B_0 *)/WIDTH*,

B_e is denoted by *(NUM(start bit) + B_0) MOD WIDTH,*
and

L_e is denoted by *NUM(length)*

where *WIDTH* is the numer of bits in a word.

Step

Let element$_i$ be

- the element of the lowest index, if i=0
- the element of the index which is the successor of the index for the element$_n$ where n = i-1 otherwise

Let i_0 be the number of elements preceding element$_0$ in its positioning pattern. Given a step attribute of the form

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

the *pos* of an individual element with respect to the beginning of the positioning pattern of element$_0$ is determined as follows:

the *pos* of element$_i$ is:
*POS (NUM(pattern size) * ((i + i_0) /DENS)*
+ RBPOS$_i$/WIDTH, RBPOS$_i$ MOD WIDTH, length)
for 1 ≤ i ≤ 'number of elements'
where *RBPOS$_i$* is
*NUM(word number) * WIDTH + NUM(start bit)*
*+ NUM(step size) * ((i + i_0) MOD DENS) ,*
and *DENS* is
*(NUM(pattern size) * WIDTH)/ NUM(step size) ,*
and *WIDTH* is the number of bits in a word.

Defaults

The notation:
POS (<word number> , <start bit> : <end bit>)
is semantically equivalent to:
POS (<word number> , <start bit> ,
NUM(end bit) - NUM(start bit) + 1)

The notation:
POS (<word number> , <start bit>)
is semantically equivalent to:
POS (<word number> , <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 number>)
is semantically equivalent to:

*POS (<word number> , 0 , WSIZE * WIDTH)*
where *WSIZE* is the size of the mode of the component for which the *pos* is specified.

The notation:
STEP (<pos> , <step size>)
is semantically equivalent to
STEP (<pos> , <step size> , PSIZE)
where *PSIZE* is the smallest integer such that
*PSIZE * WIDTH ≥ NUM(step size)*

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 a mapped array mode the element layout of the mode determines the (language) referability of its sub-locations (including sub-arrays, array slices) as follows:

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

For any location of a mapped 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 (language) 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 and not a mapped mode otherwise.

Each of the *integer literal expression* occurrences must deliver a non-negative value. In addition, *length*, *step size* and *pattern size* must deliver a non-zero value, and *start bit* and *end bit* must deliver a value less than *WIDTH* where *WIDTH* is the number of bits in a word (implementation defined). Moreover, *start bit* must deliver a value not greater than the value delivered by *end bit*.

For each field name of a mapped structure mode, the *length* in its *field layout* must not be less than the minimum number of bits which need to be occupied by the field.

For each mapped array mode, the *length* in the *pos* in its *element layout* must not be less than the minimum number of bits which is needed to be occupied by the elements. In addition, for any element layout the following conditions must be fulfilled:

- *NUM(step size) ≥ NUM(length)*

- *(NUM(pattern size) * WIDTH) MOD NUM(step size)*
  *≥ NUM(word number) * WIDTH + NUM(start bit)*

Consistency and feasibility

Consistency:

No component of an array or structure object may be specified to occupy any bits occupied by another component of the same object except in the case of two variant field names defined in the same *alternative fields* occurrence; however, in the latter case the variant fields 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 which themselves have referable components.

examples:

17.5	*PACK*	*(1.1)*
19.11	*POS(1,0:15)*	*(4.2)*

3.11 DYNAMIC MODES

3.11.1 GENERAL

A dynamic mode is a mode some properties of which are known only at run time. Dynamic modes are always parameterised modes with one or more run-time parameters. Dynamic modes have no denotation in CHILL. However, for description purposes, virtual denotations are introduced in this document. These virtual denotations are preceded by the ampersand symbol (&) to distinguish them from actual notations which may appear in a CHILL program text.

3.11.2 *DYNAMIC STRING MODES*

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

semantics: A dynamic string mode is a parameterised string mode with statically unknown length. The dynamic string length is the value delivered by the *integer expression*.

static properties:

- The dynamic string mode is a bit (character) string mode if and only if the *origin string mode name* is a bit (character) string mode.

dynamic properties:

- A dynamic string mode has a dynamic length, which is the value delivered by *NUM(integer expression)*.

3.11.3 *DYNAMIC ARRAY MODES*

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

semantics: A dynamic array mode is a parameterised array mode with statically unknown upper bound. The lower bound, index mode and element mode are statically known, the dynamic upper bound is the value delivered by the *discrete expression*.

static properties:

- A dynamic array mode has an index mode, element mode, element layout and lower bound attached, which are the index mode, element mode, element layout and lower bound of the *origin array mode name*.

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 *NUM(upper bound) - NUM(lower bound) + 1*

3.11.4 *DYNAMIC PARAMETERISED STRUCTURE MODES*

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

semantics: A dynamic parameterised structure mode is a parameterised structure mode with statically unknown parameters. The composition of the structure mode can only be determined dynamically from the list of values delivered by *expression list*.

static properties:

- A dynamic parameterised structure mode has a unique origin variant structure mode attached, which is the mode denoted by the *origin variant structure mode name*.

- A dynamic parameterised structure mode is tagged if and only if its origin variant structure mode is a tagged variant structure mode, otherwise it is tag-less.

- The set of field names (fixed field names, tag field names, variant field names) of a dynamic parameterised structure mode is the set of field names (fixed field names, tag field names, variant field names) of its origin variant structure mode.

dynamic properties:

- A dynamic parameterised structure mode has a list of values attached, which is the list of values delivered by the expressions in the *expression list*.

4.0 LOCATIONS AND THEIR ACCESSES

4.1 DECLARATIONS

4.1.1 GENERAL

syntax:

```
<declaration statement> ::=                                    (1)
     DCL <declaration> {,<declaration>}*;                      (1.1)

<declaration> ::=                                              (2)
     <location declaration>                                    (2.1)
   | <loc-identity declaration>                                (2.2)
   | <based declaration>                                       (2.3)
```

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)
6.10    d, m, y INT                                            (2.1)
11.34   starting_square LOC := b(m.lin_1)(m.col_1)             (2.2)
```

4.1.2 LOCATION DECLARATIONS

syntax:

```
<location declaration> ::=                                     (1)
     <name list> <mode> [STATIC] [<initialisation>]            (1.1)

<initialisation> ::=                                           (2)
     <reach-bound initialisation>                              (2.1)
   | <lifetime-bound initialisation>                           (2.2)

<reach-bound initialisation> ::=                               (3)
     <assignment symbol> <value> [<handler>]                   (3.1)

<lifetime-bound initialisation> ::=                            (4)
     INIT <assignment symbol> <constant value>                 (4.1)
```

semantics: A location declaration creates as many locations as there are names specified in the *name list*.

With *reach-bound initialisation*, the *value* is evaluated each time the reach in which the declaration is placed is entered (see section 7.2) and the delivered value is assigned to the

location(s). Before the *value* is evaluated the location contains an undefined value (except when a mode with the tagged parameterised property or with the synchronisation property is specified; see below).

With *lifetime-bound initialisation*, 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 sections 7.2 and 7.9).

Specifying no *initialisation* is semantically equivalent to the specification of a *lifetime-bound initialisation* with the *undefined value* (see section 5.3). However, if the *mode* has the tagged parameterised property, the tag field sub-locations of the location are initialised with the corresponding value of the associated parameterised structure mode.

The meaning of the *undefined value* as initialisation for a synchronisation location is that the created event and/or buffer sub-locations are automatically initialised to "empty", i.e. no delayed processes are attached to the event or buffer, nor are there messages in the buffer.

The semantics of *STATIC* and *handler* can be found in section 7.9 and chapter 10, respectively.

static properties: Names declared in a location declaration are location names. The mode attached to the location name is the *mode* specified in the *location declaration*. A location name is (language) referable.

static conditions: The class of the *value* or *constant value* must be compatible with the *mode*.

If the *mode* has the read-only property, *initialisation* must be specified. If the *mode* has the synchronisation property, *reach-bound initialisation* must not be specified.

dynamic conditions: In the case of *reach-bound initialisation*, the assignment conditions of *value* with respect to the *mode* apply (see section 6.2).

examples:

```
5.8     k2, x, w, t, s, r BOOL                    (1.1)
6.9     := julian_day_number                      (3.1)
8.4     INIT := ['A':'Z']                         (4.1)
```

4.1.3 *LOC-IDENTITY DECLARATIONS*

syntax:

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

semantics: A loc-identity declaration creates as many accesses to the specified static mode location as there are names specified in the *name list*.

If the *static mode location* is evaluated dynamically, this evaluation is done each time that 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 sections 7.2 and 7.9).

static properties: Names declared in a loc-identity declaration are loc-identity names. The mode attached to a loc-identity name is the *mode* specified in the *loc-identity declaration*. A loc-identity name is (language) referable if and only if the specified *static mode location* is (language) referable.

static conditions: The specified *mode* must be read-compatible with the mode of the *static mode location*.

examples:

11.34 starting square LOC := b(m.lin_1)(m.col_1) (1.1)

4.1.4 *BASED DECLARATIONS*

syntax:

<based declaration> ::= (1)
<name list> <mode> BASED
[(<bound or free reference location name>)] (1.1)

derived syntax: A *based declaration* without a *bound or free reference location name*, is derived syntax for a synmode definition statement. E.g.
DCL I INT BASED;
is derived from:
SYNMODE I = INT;
The declared names are synmode names, synonymous with the specified *mode*.

semantics: A based declaration (with *bound or free reference location name*) specifies as many accesses as there are names in the *name list*. Names declared in a based declaration serve as an alternative way of accessing a location by dereferencing a

reference value. This reference value is contained in the location specified by the *bound or free reference location name*. This dereferencing operation is made each time and only when an access is made via a declared based name.

static properties: The names declared in a *based declaration* are based names. The mode attached to a based name is the *mode* specified in the *based declaration*. A based name is (language) referable.

static conditions: If the mode of the *bound or free reference location name* is a bound reference mode, the specified *mode* must be read-compatible with the referenced mode of the mode of the *bound or free reference location name*.

examples:

```
19.9    1 X BASED (P),
          2 I INFO POS(0,8:31),
          2 PREV PTR POS(1,0:15),
          2 NEXT PTR POS(1,16:31)                    (1,1)
```

4.2 LOCATIONS

4.2.1 GENERAL

syntax:

```
<location> ::=                                       (1)
     <static mode location>                          (1.1)
   | <dynamic mode location>                         (1.2)

<static mode location> ::=                           (2)
     <access name>                                   (2.1)
   | <dereferenced bound reference>                  (2.2)
   | <dereferenced free reference>                   (2.3)
   | <string element>                                (2.4)
   | <sub string>                                    (2.5)
   | <array element>                                 (2.6)
   | <sub-array>                                     (2.7)
   | <structure field>                               (2.8)
   | <location procedure call>                       (2.9)
   | <location built-in routine call>                (2.10)
   | <location conversion>                           (2.11)

<dynamic mode location> ::=                          (3)
     <string slice>                                  (3.1)
   | <array slice>                                   (3.2)
   | <dereferenced row>                              (3.3)
```

semantics: A location is an object that can contain values. A location is either denoted by a *static mode location*, i.e. its mode is statically determinable, or by a *dynamic mode location*, i.e. part of the mode information can only be obtained dynamically. Locations have to be accessed to store or obtain a value.

static properties: A *static mode location* has a static mode attached. For descriptive purposes only, a virtual dynamic mode will be attached to each *dynamic mode location* (see section 3.1). In the case of dynamic mode locations the required compatibility checks can be completely performed only at run time. Check failure of the dynamic part will cause either the *RANGEFAIL* or the *TAGFAIL* exception.

4.2.2 ACCESS NAMES

syntax:

```
<access name> ::=                                   (1)
    <location name>                                 (1.1)
  | <loc-identity name>                             (1.2)
  | <based name>                                    (1.3)
  | <location enumeration name>                     (1.4)
  | <location do-with name>                         (1.5)
```

semantics: An access name is an access to 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 based name, i.e. a name declared in a *based declaration*;
- a location enumeration name, i.e. a *loop counter* in *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 tag-less variant structure location, the semantics are implementation defined.

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

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

dynamic conditions: A *loc-identity name* must not denote an undefined location.

When accessing via *based name*, the same dynamic conditions hold as when dereferencing the *bound or free reference location name* in the associated *based declaration* (see sections 4.2.3 and 4.2.4).

Accessing via a *location do-with name* causes a *TAGFAIL* exception if the denoted location is a variant field of:

- a tagged variant structure mode location and the associated tag field value(s) indicate(s) that the field does not exist;
- a dynamic parameterised structure mode location and the associated list of values indicates that the field does not exist.

examples:

4.11	*a*	*(1.1)*
11.38	*starting*	*(1.2)*
19.14	*X*	*(1.3)*
15.25	*EACH*	*(1.4)*
5.11	*c1*	*(1.5)*

4.2.3 DEREFERENCED BOUND REFERENCES

syntax:

<dereferenced bound reference> ::= *(1)*
<bound reference expression> -> [<mode name>] *(1.1)*

semantics: The location obtained by dereferencing a bound reference value is that which is referenced by the bound reference value.

static properties: The mode attached to the *dereferenced bound reference* is the *mode name* if one is specified, otherwise the referenced mode of the mode of the *bound reference expression*. A *dereferenced bound reference* is (language) referable.

static conditions: The *bound reference expression* 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 expression*.

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

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

examples:

10.49 p-> *(1.1)*

4.2.4 DEREFERENCED FREE REFERENCES

syntax:

<dereferenced free reference> ::= *(1)*
<free reference expression> -> <mode name> *(1.1)*

semantics: The location obtained by dereferencing a free reference value is that which is referenced by the free reference value.

static properties: The mode attached to the *dereferenced free reference* is the *mode name*. A *dereferenced free reference* is (language) referable.

static conditions: The *free reference expression* must be **strong**.

dynamic conditions: The *free reference expression* must not deliver a value obtained by referencing a non-referable location (see section 5.2.13). The lifetime of the referenced location must not have ended.

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

The *MODEFAIL* exception occurs if the *mode name* is not read-compatible with the mode of the referenced location.

4.2.5 STRING ELEMENTS

syntax:

<string element> ::= *(1)*
<string location> (<position>) *(1.1)*

derived syntax: A *string element* is derived syntax for a substring of length 1 (see section 4.2.6). E.g.
<string location> (<position>)
is derived from:
<string location> (<position> UP 1)

examples:

```
18.16   string->(i)                                          (1.1)
```

4.2.6 SUBSTRINGS

syntax:

```
<substring> ::=                                              (1)
     <string location> ( <left element> : <right element> ) (1.1)
   | <string location> ( <position> UP <string length> )     (1.2)

<left element> ::=                                           (2)
     <integer literal expression>                            (2.1)

<right element> ::=                                          (3)
     <integer literal expression>                            (3.1)

<position> ::=                                               (4)
     <integer expression>                                    (4.1)
```

semantics: A substring delivers a string location which is a substring of the specified string location.

static properties: The mode attached to a *substring* is a parameterised string mode, constructed as:
&name (substring length)
where *&name* is a virtual synmode name synonymous with the (possibly dynamic) mode of the *string location*, and where *substring length* is either *string length* or
NUM(right element) - NUM(left element) + 1

static conditions: The *left element*, *right element* and *length* must deliver non-negative integer values such that the following relations hold:

1. *NUM(left element)* $\leq$ *NUM(right element)*

2. *NUM(right element)* $\leq$ *L-1*

3. *1* $\leq$ *NUM(string length)* $\leq$ *L*

where *L* is the string length of the *string location*. (If the *string location* is a *dynamic mode location*, the relations 2. and 3. can only be checked at run time; see below.)

dynamic conditions: The *RANGEFAIL* exception occurs if any of the relations 2. or 3. above does not hold in the case of a dynamic mode *string location*, or if any of the following relations hold:

1. *NUM(position) < 0*

2. *NUM(position) + NUM(string length) > L*

where *L* is the string length of the mode of the *string location*.

examples:

18.23 string->(scanstart UP 10) *(1.2)*

4.2.7 ARRAY ELEMENTS

syntax:

<array element> ::= *(1)*
<array location> (<expression list>) *(1.1)*

<expression list> ::= *(2)*
*<expression> {, <expression>}** *(2.1)*

derived syntax: The notation: *(<expression list>)* is derived syntax for:
*(<expression>) {(<expression>)}**
where there are as many parenthesised 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 an element of the specified array location.

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

An *array element* is (language) 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 any of the following relations hold:

1. *expression < L*

2. *expression > U*

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

examples:

```
11.34    b(m.lin_1)(m.col_2)                                    (1.1)
```

4.2.8 SUB-ARRAYS

syntax

```
<sub-array> ::=                                                 (1)
     <array location>( <lower element> : <upper element> )      (1.1)
   | <array location>
      ( <integer expression> UP <array length> )                (1.2)

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

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

<array length> ::=                                              (4)
     <integer literal expression>                               (4.1)
```

semantics: A sub-array delivers a (sub) array location which is the part of the specified array location indicated by the *lower element* and *upper element*, or *integer expression* and *array length*. The lower bound of the sub-array is equal to the lower bound of the specified array; the upper bound is determined from the specified expressions.

static properties: The mode attached to a *sub-array* is a parameterised array mode defined as follows:
&name(upper index)
where *&name* is a virtual synmode name synonymouth with the (possibly dynamic) mode of the *array location* and *upper index* is either *L + array length - 1*, where *L* is the lower bound of the array mode of the *array location*, or *lit*, where *lit* is a literal whose class is compatible with the classes of *lower element* and *upper element* such that:
NUM(lit) = NUM(L) + NUM(upper element) - NUM(lower element).
A *sub-array* is (language) 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 *integer expression* and *array length* must be compatible with the index mode of the mode of the *array location*.

The *lower element*, *upper element*, and *array length* must deliver values such that the following relations hold:

1. *L* ≤ *lower element* ≤ *upper element*

2. 1 ≤ *array length*

3. *upper element* ≤ *U*

4. *array length* ≤ *U* - *L* + 1

Where *L* and *U* are respectively the lower bound and upper bound of the mode of the *array location*. (If the *array location* is a *dynamic mode location*, relations 3. and 4. can only be checked at run time; see below.)

dynamic conditions: The *RANGEFAIL* exception occurs if any of the relations 3. and 4. above does not hold for a dynamic mode *array location*, or if any of the following relations hold:

1. *L* > *integer expression*

2. *integer expression* + *array length* - 1 > *U*

Where *L* and *U* are the lower bound and upper bound of the array mode of the *array location*, respectively.

4.2.9 *STRUCTURE FIELDS*

syntax:

<structure field> ::= (1)
<structure location> . *<field name>* (1.1)

semantics: A structure field delivers a (sub)location which is a field of the specified structure location.

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 (language) referable if the *field name* is (language) referable.

static conditions: The *field name* must be a name from the set of field names of the mode of the *structure location*.

dynamic conditions: The *TAGFAIL* exception occurs if the *structure location* denotes:

- a tagged variant structure mode location and the associated tag field value(s) indicate(s) that the field does not exist;

- a dynamic parameterised structure mode location and the associated list of values indicates that the field does not exist.

examples:

```
10.52    last->.info                                    (1.1)
```

4.2.10 LOCATION PROCEDURE CALLS

syntax:

```
<location procedure call> ::=                           (1)
    <location procedure call>                           (1.1)
```

semantics: A location is delivered as the result of a location procedure call.

static properties: The mode attached to a *location procedure call* is the mode of 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.

4.2.11 LOCATION BUILT-IN ROUTINE CALLS

syntax:

```
<location built-in routine call> ::=                    (1)
    <implementation location built-in routine call>     (1.1)
```

semantics: A location is delivered as the result of a implementation location built-in routine call.

static properties: The mode attached to the *location built-in routine call* is the result mode of the *implementation location built-in routine call*.

dynamic conditions: The *implementation location built-in routine call* must not deliver an undefined location and the lifetime of the delivered location must not have ended.

4.2.12 LOCATION CONVERSIONS

syntax:

```
<location conversion> ::=                               (1)
    <mode name>( <static mode location> )               (1.1)
```

semantics: A location conversion overrides the CHILL mode checking and compatibility rules. It explicitly attaches a mode to the specified static mode location.

The precise dynamic semantics of a location conversion are implementation defined.

static properties: The mode of a *location conversion* is the *mode name*.

static conditions: The *static mode location* must be referable.

The following relation must hold:
SIZE(mode name) = SIZE(static mode location)

4.2.13 STRING SLICES

syntax:

```
<string slice> ::=                                          (1)
     <string location>( <start> : <end> )                   (1.1)

<start> ::=                                                 (2)
     <integer expression>                                   (2.1)

<end> ::=                                                   (3)
     <integer expression>                                   (3.1)
```

N.B. If both *start* and *end* are an *integer literal expression*, the syntactic construct is ambiguous and will be interpreted as a *substring*.

semantics: A string slice delivers a dynamic mode string location, i.e. a string with statically unknown string length.

static properties: The dynamic mode attached to a *string slice* is a dynamic parameterised string mode, formed in the same way as for a *substring* (see section 4.2.6), but with a dynamic *string length* parameter formed by:
NUM(end) - NUM(start) + 1

dynamic conditions: The *RANGEFAIL* exception occurs if any of the following relations hold:

1. *NUM(start) > NUM(end)*

2. *NUM(start) < 0*

3. *NUM(end)* $\geq$ *L*

where *L* is the (possibly dynamic) length of the string mode of the *string location*.

examples:

```
18.26   blanks(count:9)                                    (1.1)
```

4.2.14 *ARRAY SLICES*

syntax:

```
<array slice> ::=                                          (1)
    <array location>( <first> : <last> )                   (1.1)

<first> ::=                                                (2)
    <expression>                                           (2.1)

<last> ::=                                                 (3)
    <expression>                                           (3.1)
```

N.B. If both *first* and *last* are a *literal expression*, the syntactic construct is ambiguous and will be interpreted as a sub-array.

semantics: An array slice delivers a dynamic mode array location, i.e. an array with a statically unknown upper bound.

static properties: The dynamic mode attached to an *array slice* is a dynamic parameterised array mode formed in the same way as for a *sub-array* (see section 4.2.8) but with a dynamic *upper index* parameter formed by *exp*, where *exp* is an expression whose class is compatible with the classes of *first* and *last* and such that:
NUM(exp) = NUM(L) + NUM(last) - NUM(first)
where *L* is the lower bound of the mode of the *array location*.

An *array slice* is (language) referable if the element layout of the mode of the *array location* is *NOPACK*.

static conditions: The classes of *first* and *last* must be compatible with the index mode of the mode of the *array location*.

dynamic conditions: The *RANGEFAIL* exception occurs if any of the following relations hold:

1. *first > last*
2. *first < L*
3. *last > U*

where *L* and *U* denote respectively the lower bound and the (possibly dynamic) upper bound of the mode of the *array location*.

examples:

17.27 res(0:count-1) (1.1)

4.2.15 DEREFERENCED ROWS

syntax:

<dereferenced row> ::= (1)
<row expression> -> (1.1)

semantics: A dereferenced ROW delivers the dynamic mode location that is referenced by the ROW value.

static properties: The dynamic mode attached to the *dereferenced row* is constructed as follows:
&origin mode name(<parameter> {,<parameter>})*
where *origin mode name* is a virtual synmode name synonymous with the referenced origin mode of the mode of the (strong) *row expression*, and where the parameters are, depending on the referenced origin mode:

- the dynamic 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 parameterised structure location, in the case of a variant structure mode.

A *dereferenced row* is (language) referable.

static conditions: The *row expression* must be strong.

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

The *EMPTY* exception occurs if the *row expression* delivers *NULL*.

examples:

8.10 input-> (1.1)

5.0 VALUES AND THEIR OPERATIONS

5.1 SYNONYM DEFINITIONS

syntax:

```
<synonym definition statement> ::=                                  (1)
     SYN <synonym definition> {,<synonym definition>}*;             (1.1)

<synonym definition> ::=                                            (2)
     <name list> [<mode>] = <constant value>                        (2.1)
```

derived syntax: A *synonym definition* where *name list* consists of more than one name, is derived from several *synonym definition* occurrences, one for each name, with the same *constant value* and *mode* if present. E.g. `SYN I,J=3;` is derived from `SYN I=3, J=3;`

semantics: A synonym definition defines a name to denote the specified constant value.

static properties: A name defined in a synonym definition is a synonym name.

The class of the synonym name is, if a *mode* is specified, the M-value class, where M is the *mode*, otherwise the class of the *constant value*.

A synonym name is undefined if and only if the *constant value* is an *undefined value* (see section 5.3.1).

A synonym name is literal if and only if the *constant value* is a *literal expression*.

static conditions: If a *mode* is specified, it must be compatible with the class of the *constant value* and the value delivered by the *constant value* must be one of the values defined by the *mode*.

Synonym definitions must not be recursive nor mutually recursive via other synonym definitions or mode definitions, i.e. no set of recursive definitions may contain synonym definitions (see section 3.2.1).

examples:

```
1.14    SYN neutral_for_add = 0,                                    (1.1)
           neutral_for_mult = 1;
2.17    neutral_for_add fraction = [0,1]                            (2.1)
```

5.2 PRIMITIVE VALUE

5.2.1 GENERAL

syntax:

```
<primitive value> ::=                                   (1)
      <location contents>                               (1.1)
    | <value name>                                      (1.2)
    | <literal>                                         (1.3)
    | <tuple>                                           (1.4)
    | <value string element>                            (1.5)
    | <value substring>                                 (1.6)
    | <value string slice>                              (1.7)
    | <value array element>                             (1.8)
    | <value sub-array>                                 (1.9)
    | <value array slice>                               (1.10)
    | <value structure field>                           (1.11)
    | <referenced location>                             (1.12)
    | <expression conversion>                           (1.13)
    | <value procedure call>                            (1.14)
    | <value built-in routine call>                     (1.15)
    | <start expression>                                (1.16)
    | <receive expression>                              (1.17)
    | <zero-adic operator>                              (1.18)
```

semantics: A primitive value is the basic consituent of an expression. Some primitive values (location contents of a dynamic mode location, some tuples, value array slices, value string slices) 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 a *constant primitive value* if and only if it is a *constant value name*, *literal*, *constant tuple*, *constant referenced location*, *constant expression conversion* or *constant value built-in routine call*.

A *primitive value* is a *literal primitive value*, if and only if it is a *literal value name*, a *discrete literal* or a *literal value built-in routine call*.

5.2.2 *LOCATION CONTENTS*

syntax:

```
<location contents> ::=                                   (1)
    <location>                                            (1.1)
```

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 synchronisation property.

dynamic conditions: The delivered value must not be undefined (see section 5.3.1).

examples:

```
3.6     c2.im                                             (1.1)
```

5.2.3 *VALUE NAMES*

syntax:

```
<value name> ::=                                          (1)
    <synonym name>                                        (1.1)
  | <value enumeration name>                              (1.2)
  | <value do-with name>                                  (1.3)
  | <value receive name>                                  (1.4)
```

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 *loop counter* in *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*.

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*, respectively.

A *value name* is constant (literal) if and only if it is a *synonym name* (*literal synonym name*).

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 of:

- a tagged variant structure mode value and the associated tag field(s) indicate(s) that the denoted field does not exist;
- a dynamic parameterised structure mode value and the associated list of values indicates that the denoted field does not exist.

examples:

```
10.9    max                                   (1.1)
8.8     i                                     (1.2)
15.45   THIS_COUNTER                          (1.4)
```

5.2.4 *LITERALS*

5.2.4.1 General

syntax:

```
<literal> ::=                                 (1)
      <integer literal>                       (1.1)
    | <boolean literal>                       (1.2)
    | <set literal>                           (1.3)
    | <emptiness literal>                     (1.4)
    | <procedure literal>                     (1.5)
    | <character string literal>              (1.6)
    | <bit string literal>                    (1.7)
```

semantics: A literal delivers a constant value which is known at compile time.

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 *set literal*, a *character string literal* of length 1, or a *bit string literal* of length 1.

5.2.4.2 Integer literals

syntax:

```
<integer literal> ::=                                          (1)
      <decimal integer literal>                                (1.1)
    | <binary integer literal>                                 (1.2)
    | <octal integer literal>                                  (1.3)
    | <hexadecimal integer literal>                            (1.4)

<decimal integer literal> ::=                                  (2)
      [D'] {<digit> | _}*                                      (2.1)

<binary integer literal> ::=                                   (3)
      B' {0 | 1 | _}*                                          (3.1)

<octal integer literal> ::=                                    (4)
      O' {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | _}*                 (4.1)

<hexadecimal integer literal> ::=                              (5)
      H' {<hexadecimal digit> | _}*                            (5.1)

<digit> ::=                                                    (6)
      0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9                  (6.1)

<hexadecimal digit> ::=                                        (7)
      <digit> | A | B | C | D | E | F                          (7.1)
```

semantics: An integer literal delivers an integer value. The usual decimal notation is provided as well as binary, octal, hexadecimal and explicit decimal. The underline symbol (_) is not significant, i.e. it serves only for readability and it does not influence the denoted value.

static properties: The class of an *integer literal* is the *INT*-derived class.

static conditions: The string following the apostrophe (') and the whole *integer literal* must not consist solely of underline symbols.

examples:

```
6.11      1_721_119                                            (1.1)
---       D'1_721_119                                          (1.1)
---       B'101011_110100                                      (1.2)
---       O'53_64                                              (1.3)
---       H'AF4                                                (1.4)
```

5.2.4.3 Boolean literals

syntax:

<boolean literal> ::= (1)
FALSE | TRUE (1.1)

semantics: A boolean literal delivers a boolean value.

static properties: The class of a *boolean literal* is the *BOOL*-derived class.

examples:

5.46 FALSE (1.1)

5.2.4.4 Set literals

syntax:

<set literal> ::= (1)
<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-derived class, where M is the set mode (in the given context) which has the specified *set element name* as a set element name.

examples:

6.51 dec (1.1)
11.89 king (1.1)

5.2.4.5 Emptiness literal

syntax:

<emptiness literal> ::= (1)
NULL (1.1)

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.

examples:

10.40 *NULL* (1.1)

5.2.4.6 Procedure literals

syntax:

<procedure literal> ::= (1)
    <general procedure name> (1.1)

semantics: A procedure literal delivers a general procedure value. A procedure literal is a name defined in a procedure definition or entry definition (see section 7.4).

static properties: The class of the *procedure literal* is the M-derived class, where M is the mode of the *general procedure name*.

5.2.4.7 Character string literals

syntax:

<character string literal> ::= (1)
    ' {<non-apostrophe character> | <apostrophe>}'* (1.1)
  | *C' {<hexadecimal digit> <hexadecimal digit>}*'* (1.2)

<character> ::= (2)
    <letter> (2.1)
  | *<digit>* (2.2)
  | *<symbol>* (2.3)
  | *<space>* (2.4)

<letter> ::= (3)
    A | B | C | D | E | F | G | H | I | J | K | L | M (3.1)
  | *N | O | P | Q | R | S | T | U | V | W | X | Y | Z* (3.2)

<symbol> ::= (4)
    *_ | ' | (|) | * | + | , | - | . | / | : | ; | < | = | >* (4.1)

<space> ::= (5)
    SP (5.1)

<apostrophe> ::= (6)
    '' (6.1)

Note: SP denotes the character "space"; see Appendix A1.

semantics: A character string literal delivers a character string value, which may be of length 0. A character string literal of length 1 may serve as a character value. To represent the character

apostrophe (') within a character string literal it has to be written twice (''). The above mentioned characters constitute the minimum printable character set that must be provided. An implementation may allow any printable character within a character string literal that is in the CCITT alphabet no. 5 (see Appendix A1). Apart from the printable representation, the hexadecimal representation may be used. Each hexadecimal digit pair denotes that character value whose representation corresponds to the given hexadecimal number (see Appendix A1).

static properties: The length of a *character string literal* is either the number of *non-apostrophe character* and *apostrophe* occurrences, or the number of *hexadecimal digit* pairs.

The class of a *character string literal* is the *CHAR(n)*-derived class, where *n* is the length of the *character string literal*.

examples:

8.18 *'A-B<ZAA9K'''* *(1.1)*
8.18 *''* *(6.1)*

5.2.4.8 Bit string literals

syntax:

<bit string literal> ::= *(1)*
<binary bit string literal> *(1.1)*
| *<octal bit string literal>* *(1.2)*
| *<hexadecimal bit string literal>* *(1.3)*

<binary bit string literal> ::= *(2)*
B' { 0 | 1 | _ }'* *(2.1)*

<octal bit string literal> ::= *(3)*
O' {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | _ }'* *(3.1)*

<hexadecimal bit string literal> ::= *(4)*
H' {<hexadecimal digit> | _ }'* *(4.1)*

semantics: A bit string literal delivers a bit string value which may be of length 0. Binary, octal or hexadecimal notations may be used. The underline symbol (_) is insignificant, i.e. it serves only for readability and does not influence the indicated value.

static properties: The length of a *bit string literal* is either the number of *0* and *1* occurrences after *B'*, or three times the number of *0*, *1*, *2*, *3*, *4*, *5*, *6* and *7* occurrences after *O'*, or four times the number of *hexadecimal digit* occurrences after *H'*.

The class of a *bit string literal* is the *BIT(n)*-derived class, where *n* is the length of the *bit string literal*.

examples:

```
---     B'101011_110100'                                    (1.1)
---     O'53_64'                                            (1.2)
---     H'AF4'                                              (1.3)
```

5.2.5 TUPLES

syntax:

```
<tuple> ::=                                                 (1)
     [<mode name>] (:
     {<powerset tuple> | <array tuple> | <structure tuple>}
     :)                                                     (1.1)

<powerset tuple> ::=                                        (2)
     [{<expression> | <range>}
     {, {<expression> | <range>}}*]                         (2.1)

<range> ::=                                                 (3)
     <expression> : <expression>                            (3.1)

<array tuple> ::=                                           (4)
     <unlabelled array tuple>                               (4.1)
   | <labelled array tuple>                                 (4.2)

<unlabelled array tuple> ::=                                (5)
     <value> {,<value>}*                                    (5.1)

<labelled array tuple> ::=                                  (6)
     <case label list> : <value>
     {, <case label list> : <value>}*                       (6.1)

<structure tuple> ::=                                       (7)
     <unlabelled structure tuple>                           (7.1)
   | <labelled structure tuple>                             (7.2)

<unlabelled structure tuple> ::=                            (8)
     <value> {,<value>}*                                    (8.1)

<labelled structure tuple> ::=                              (9)
     <field name list> : <value>
     {, <field name list> : <value>}*                       (9.1)

<field name list> ::=                                       (10)
     .<field name> {, .<field name>}*                       (10.1)
```

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 section 9.1.4).

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 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 mixed 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 the following list:

- if the *tuple* is the *value* or *constant value* in an *initialisation* in a *location declaration*, then M is the *mode* in the *location declaration*;
- if the *tuple* is the righthand side *value* in a *single assignment action*, then M is the (possibly dynamic) mode of the lefthand 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 an *actual parameter* in a *procedure call*, 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 result mode of the procedure name of the *result action* or *return action* (see section 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.

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 deleted 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. 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*.

In the case of an *unlabelled array tuple*:

3. There must be as many occurrences of *value* as the number of elements of the array mode of the *tuple*.

In the case of a *labelled array tuple*:

4. The case selection conditions must hold for the list of *case label list* occurrences (see section 9.1.3). The resulting class must be compatible with the index mode of the mode of the *tuple* .

5. The value delivered by each *literal expression* in each *case label list* and the values defined by each *mode name* in each *case label list* must lie between the lower bound and upper bound (bounds included) of the mode of the *tuple*.

6. In an *unlabelled array tuple*, at least one *value* occurrence must be an *expression*; in a *labelled array tuple*, at least one *value* occurrence following a *case label list* which is not *(ELSE)* must be an *expression* (see section 5.3.1).

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 within the bounds of the element mode (bounds included), 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 section 9.2.7).

In the case of an *unlabelled structure tuple*:

- If the mode of the *tuple* is neither a variant structure mode nor a parameterised 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 parameterised structure mode then:

  5. Each *value* specified for a tag field must be a *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 *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 parameterised 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 parameterised 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 a *field name list* and in the same order as in the mode of the *tuple*.

  10. The class of each *value* must be compatible with the mode of the field name specified in the *field name list* labelling that *value*.

- If the mode of the *tuple* is a tagged variant structure mode or a tagged parameterised structure mode, then:

  11. Each *value* that is specified for a tag field must be a *literal expression*,

  12. Only field names corresponding to fields indicated as existing by the value(s) delivered by the *literal expression* occurrences specified for the tag fields may be specified and all of them must be specified in the same order as in the mode of the *tuple*.

  13. The class of each *value* must be compatible with the mode of the 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 parameterised structure mode then:

  14. Field names mentioned in *field name list*, which are defined in the same *alternative fields*, must be all defined in the same *variant alternative* or defined after *ELSE*. All the field names of a selected variant alternative or defined after *ELSE*, must be mentioned once and only once in the same order as in the mode of the *tuple*.

  15. The class of each *value* must be compatible with the mode of any field name specified in the *field name list* in front of that *value*.

16. If the mode of the *tuple* is a tagged parameterised structure mode, the list of values delivered by the *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 within the bounds of the mode of the field (bounds included), unless it is an *undefined value*.

18. At least one *value* occurrence must be an *expression*.

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 section 6.2.3) apply (refer conditions a2, a3, 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 fail.

If the *tuple* has a dynamic parameterised structure mode, the *TAGFAIL* exception occurs if the check c16 fails.

examples:

9.5	*number_list[]*	(1.1)
9.6	*[2:max]*	(2.1)
8.24	*[('A'):3,('B','K','Z'):1,(ELSE):0]*	(6.1)
17.5	*[(*):' ']*	(6.1)
12.28	*(:NULL,NULL,536:)*	(7.1)
11.16	*[.status:occupied,.p:[white,rook]]*	(9.1)

5.2.6 *VALUE STRING ELEMENTS*

syntax:

<value string element> ::= (1)
<string expression>(<position>) (1.1)

derived syntax: A *value string element* is derived syntax for a *value substring* of length 1 (see section 5.2.7). I.e.
<string expression>(<position>)
is derived from:
<string expression> (<position> UP 1)

5.2.7 *VALUE SUBSTRINGS*

syntax:

<value substring> ::= (1)
<string expression>(<left element> : <right element>) (1.1)
| *<string expression>(<position> UP <string length>)* (1.2)

N.B. if the *string expression* is a *string location*, the syntactic construct is ambiguous and will be interpreted as a *substring* (see section 4.2.6).

semantics: A value substring delivers a string value which is a sub-value of the specified string value.

static properties: The class of a *value substring* is, if the *string expression* is not strong then the *CHAR(n)*-derived class or *BIT(n)*-derived class (depending on whether the *string expression* is a bit or character *string expression*) otherwise the *&name(n)*-value class, where *n* is either *<string length>* or
NUM(right element) - NUM(left element) + 1 and *&name* is a virtual synmode name, synonymous with the mode of the *string expression*.

static conditions: The *left element*, *right element* and *string length* must deliver non-negative integer values such that the following relations hold:

1. $NUM(\text{ left element }) \leq NUM(\text{ right element })$

2. $NUM(\text{ right element }) \leq L-1$

3. $1 \leq NUM(\text{ string length }) \leq L$

where L is the string length of the root mode of the class of the *string expression*. (If the *string expression* has a dynamic class, the checks 2. and 3. can be performed only at run time; see below.)

dynamic conditions: The value delivered by a *value substring* must not be undefined.

The *RANGEFAIL* exception occurs if any of the relations 2 or 3 above does not hold in the case of a *string expression* with a dynamic class, or if any of the following relations hold:

1. *NUM(positon) < 0*

2. *NUM(position) + NUM(string length) > L*

where *L* is the string length of the root mode of the class of the *string expression*.

5.2.8 *VALUE STRING SLICES*

syntax:

<value string slice> ::= (1)
<string expression>(<start> : <end>) (1.1)

N.B. if the *string expression* is a *string location* the syntactic construct is ambiguous and will be interpreted as a *string slice* (see section 4.2.13). If both *start* and *end* are an *integer literal expression*, the syntactic construct is ambiguous and will be intrepreted as a *value substring* (see section 5.2.7).

semantics: A value string slice delivers a dynamic string value, which is a sub-value of the specified string value.

static properties: The class of a *value string slice* is defined the same way as for the value substring (see section 5.2.7), but with a dynamic parameter *n* formed as:
NUM(end) - NUM(start) + 1.

dynamic conditions: The value delivered by a *value string slice* must not be undefined.

The *RANGEFAIL* exception occurs if any of the following relations hold:

1. *NUM(start) > NUM(end)*

2. *NUM(start) < 0*

3. *NUM(end) > L*

where *L* is the (possibly dynamic) length of the root mode of the class of the *string expression*.

5.2.9 *VALUE ARRAY ELEMENTS*

syntax:

<value array element> ::= (1)
<array expression> (<expression list>) (1.1)

N.B. if the *array expression* is a *array location* the syntactic construct is ambiguous and will be interpreted as a *array element* (see section 4.2.7).

derived syntax: See section 4.2.7

semantics: A value array element delivers a value which is an element of the specified array value.

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 expression*.

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

dynamic conditions: The value delivered by a *value array element* must not be undefined.

The *RANGEFAIL* exception occurs if any of the following relations hold:

1. *expression < L*

2. *expression > U*

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

5.2.10 *VALUE SUB-ARRAYS*

syntax:

<value sub-array> ::= (1)
<array expression>
(<lower element> : <upper element>) (1.1)
| *<array expression>*
(<integer expression> UP <array length>) (1.2)

N.B. if the *array expression* is an *array location* the syntactic construct is ambiguous and will be interpreted as a *sub-array* (see section 4.2.8).

semantics: A value sub-array delivers a (sub) array value which is part of the specified array value. The lower bound of the value sub-array is equal to the lower bound of the specified array value; the upper bound is determined from the specified (index) expressions.

static properties: The class of a *value sub-array* is the M-value class, where M is a parameterised array mode defined as:
&name(upper index)
where *&name* is a virtual synmode name synonymous with the (possibly dynamic) mode of the *array expression* and *upper index* is either *L + array length - 1*, where *L* is the lower bound of the mode of the *array expression*, or *lit*, where *lit* is the literal whose class is compatible with the classes of *lower element* and *upper element* such that:
NUM(lit) = NUM(L) + NUM(upper element) - NUM(lower element)

static conditions: The *array expression* must be strong The classes of *lower element* and *upper element* or *integer expression* and *array length* must be compatible with the index mode of the mode of the *array expression*. The *lower element*, *upper element* and *array length* must deliver values such that the following relations hold:

1. $L \leq$ *lower element* $\leq$ *upper element*

2. $1 \leq$ *array length*

3. *upper element* $< U$

4. *array length* $\leq U - L + 1$

where *L* and *U* are respectively the lower bound and upper bound of the array mode of the *array expression*. (If the *array expression* has a dynamic class, relations 3. and 4. can only be checked at run time; see below.)

dynamic conditions: The value delivered by a *value sub-array* must not be undefined.

The *RANGEFAIL* exception occurs if any of the relations 3. or 4. above does not hold in the case of a dynamic class, or if any of the following relations hold:

1. $L >$ *integer expression*

2. *integer expression* + *array length* $-1 > U$

where *L* and *U* are the lower bound and upper bound of the mode of the *array expression*, respectively.

5.2.11 *VALUE ARRAY SLICES*

syntax:

<value array slice> ::= (1)
<array expression> (<first> : <last>) (1.1)

N.B. if the *array expression* is an *array location*, the syntactic construct is ambiguous and will be interpreted as an *array slice* (see section 4.2.14). If both *first* and *last* are a *literal expression*, the syntactic construct is ambiguous and will be interpreted as a *value sub-array* (see section 5.2.10).

semantics: A value array slice delivers a dynamic array value, which is a sub value of the specified array value.

static properties: The class of a *value array slice* is the M-value class, where M is a dynamic parameterised array mode defined in the same way as for *value sub-array* (see section 5.2.10) but with a dynamic *upper index* parameter formed as *exp*, where *exp* is an expression whose class is compatible with the classes of *first* and *last*, and such that:
NUM(exp) = NUM(L) + NUM(last) - NUM(first)

static conditions: The *array expression* must be strong. The classes of *first* and *last* must be compatible with the index mode of the mode of the *array expression*.

dynamic conditions: The value delivered by a *value array slice* must not be undefined.

The *RANGEFAIL* exception occurs if any of the following relations hold:

1. *first > last*

2. *first < L*

3. *last > U*

where *L* and *U* denote respectively, the lower bound and (possibly dynamic) upper bound of the mode of the *array expression*.

5.2.12 *VALUE STRUCTURE FIELDS*

syntax

<value structure field> ::= (1)
<structure expression> . <field name> (1.1)

N.B. if the *structure expression* is a *structure location* the syntactic construct is ambiguous and will be interpreted as a *structure field* (see section 4.2.9).

semantics: A value structure field delivers a value which is a field of the specified structure value. If the *structure expression* 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*.

static conditions: The *structure expression* must be strong. The *field name* must be a name from the set of field names of the mode of the *structure expression*.

dynamic conditions: The value delivered by a *value structure field* must not be undefined.

The *TAGFAIL* exception occurs if the *structure expression* has

- a tagged variant structure mode and the associated tag field value(s) indicate(s) that the denoted field does not exist;
- a dynamic parameterised structure mode and the associated list of values indicates that the field does not exist.

examples:

16.49 (RECEIVE USER_BUFFER).ALLOCATOR *(1.1)*

5.2.13 *REFERENCED LOCATIONS*

syntax:

<referenced location> ::= *(1)*
-> <location> *(1.1)*

semantics: A referenced location delivers a reference to the specified location if the location is referable. If the location is not referable, it delivers a reference value which may not be dereferenced (see section 4.2.4) and which may refer to an implementation defined location.

static properties: If the *location* is referable, the class of the *referenced location* is the M-reference class, where M is the mode of the *location*. Otherwise the class of the *referenced location* is the *PTR*-derived class. A *referenced location* is constant if and only if the *location* is static.

examples:

8.23 ->c *(1.1)*

5.2.14 EXPRESSION CONVERSIONS

syntax:

```
<expression conversion> ::=                                   (1)
    <mode name> (<expression>)                                (1.1)
```

semantics: An expression conversion overrides the CHILL mode checking and compatibility rules. It explicitly attaches a mode to the expression. The precise dynamic semantics of an expression conversion are implementation defined and depend on internal representations of values.

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 *expression* must not have a dynamic class. The class of the *expression* must be compatible with at least one mode whose size is equal to the size of the *mode name*. The *mode name* must not have the synchronisation property.

5.2.15 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:

```
6.51    julian_day_number([10,dec,1979])                      (1.1)
11.68   ok_bishop(b,m)                                        (1.1)
```

5.2.16 VALUE BUILT-IN ROUTINE CALLS

syntax:

```
<value built-in routine call> ::=                             (1)
    <implementation value built-in routine call>              (1.1)
```

```
    | <CHILL value built-in routine call>

<CHILL value built-in routine call> ::=                          (2)
     NUM( <discrete expression>)                                 (2.1)
    | PRED( <discrete expression> )                              (2.2)
    | PRED( <bound reference expression> )                       (2.3)
    | SUCC( <discrete expression> )                              (2.4)
    | SUCC( <bound reference expression> )                       (2.5)
    | ABS( <integer expression> )                                (2.6)
    | ADDR( <location> )                                         (2.7)
    | CARD( <powerset expression> )                              (2.8)
    | MAX( <powerset expression> )                               (2.9)
    | MIN( <powerset expression> )                               (2.10)
    | SIZE( {<mode name> | <static mode location>} )             (2.11)
    | UPPER( {<array expression> | <string expression>} )        (2.12)
    | GETSTACK( <getstack argument> )                            (2.13)

<getstack 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)
```

derived syntax: *ADDR(<location>)* is derived syntax for *-> <location>*

semantics: A value built-in routine call is either an implementation defined built-in routine call or a CHILL defined built-in routine call, delivering a value. A CHILL value built-in routine call is an invocation of one of the CHILL defined built-in routines which delivers a value.

NUM delivers an integer value with the same internal representation as the value delivered by the discrete argument. *NUM* for set values delivers the integer value as specified by the set mode. *NUM* for character values delivers the integer value as specified by CCITT alphabet no. 5 (see Appendix A1). *NUM(TRUE)* delivers *1*, *NUM(FALSE)* delivers *0*. *NUM* for integer values delivers that integer value.

PRED and *SUCC* on discrete values deliver respectively the next lower and higher discrete value, if existing. Otherwise an exception occurs. If the discrete value is a set value from a set mode with holes, the holes are skipped (i.e. in the example in static properties of section 3.4.5, *SUCC(A)* delivers *B*, *PRED(B)* delivers *A*).

PRED and *SUCC* on bound reference values are defined only on reference values which refer to array elements. They deliver respectively the reference value refering to the array element with the next lower and higher index, if existing.

ABS is defined on integer values, delivering the absolute value of the integer value.

ADDR is an alternative notation for referencing a location.

CARD, *MAX* and *MIN* are defined on powerset values. *CARD* delivers the number of element values in the powerset value. *MAX* and *MIN* deliver respectively the greatest and smallest element value in the powerset value.

SIZE is defined on referable static mode locations and 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. In the first case, the *static mode location* will not be evaluated at run time.

UPPER is defined on (possibly dynamic) array values and string values, delivering the upper index of the array value or highest string index in the string value (i.e. string length minus 1).

GETSTACK creates a location of the specified mode on the stack (see section 7.4) and delivers a reference value for the created location. If a mode name is specified, a static mode location of that mode is created and a bound reference value is delivered. Otherwise a dynamic mode location is created, whose mode is a parameterised mode with run-time parameters as specified in the *GETSTACK* argument and a row value referring to the location is delivered.

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 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 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 argument of an *UPPER* built-in routine call is an *array expression*, the class of the *UPPER* built-in routine call is the M-value class, where M is the index mode of the array mode of the (strong) *array expression*. If the argument of an *UPPER* built-in routine call is a *string expression*, the class of the built-in routine call is the *INT*-derived class. An *UPPER* built-in routine call is constant and literal if and only if the class of the *array expression* or *string expression* is a static class.

The class of a *GETSTACK* built-in routine call is the M-reference class, where M is, depending on the *getstack argument*, either the *mode name* or a dynamic parameterised mode formed by:
&<array mode name>(<expression>) ,or
&<string mode name>(<integer expression>) , or
&<variant structure mode name>(<expression list>) ,
respectively.

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.

If the argument of a *MAX* or *MIN* built-in routine call is constant, it must not deliver the empty powerset value.

The *powerset expression* as an argument of a *CARD*, *MAX* or *MIN* built-in routine call must be strong.

The *bound reference expression* as an argument of a *PRED* or *SUCC* built-in routine call must be strong.

The *array expression* as an argument of an *UPPER* built-in routine call must be strong.

The following compatibility requirements hold for a *getstack 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*.
- There must be as many expressions in the *expression list* as there are classes in the list of classes of the *variant structure mode name* and the class of each expression must be compatible with the corresponding class in the list of classes of the *variant structure mode name*.

dynamic conditions: *PRED* and *SUCC* 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. *PRED* and *SUCC* cause the *RANGEFAIL* exception if they are applied to a bound reference value referencing the array element with the lowest or highest index. *PRED* and *SUCC* cause the *EMPTY* exception if

the *bound reference expression* delivers *NULL*.

MAX and *MIN* cause the *EMPTY* exception if they are applied to empty powerset values (i.e. containing no member values).

GETSTACK causes the *SPACEFAIL* exception if storage requirements cannot be satisfied.

GETSTACK causes the *RANGEFAIL* exception if in the *getstack argument*:

- the *expression* delivers a value which is outside 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 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.

ABS causes the *OVERFLOW* exception if the resulting value is outside the bounds defined by the root mode of the class of the argument.

examples:

9.11	*MIN(sieve)*	*(2.10)*
11.91	*PRED(col_1)*	*(2.2)*
11.91	*SUCC(col_1)*	*(2.4)*

5.2.17 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 chapter 8). Parameter passing is analogous to procedure parameter passing; however, additional actual parameters may be given with an implementation defined meaning. The start expression delivers a unique instance value identifying the created process.

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≥n), the compatibility requirements for the first n actual parameters are the same as for procedure parameter passing (see section 6.7).

dynamic conditions: The *start expression* can cause any implementation defined exception whose name is attached to the *process name* (see section 7.5).

For parameter passing, the assignment conditions of any actual value with respect to the mode of its associated formal parameter apply (see section 6.7).

The *start expression* causes the *SPACEFAIL* exception if storage requirements cannot be satisfied.

examples:

15.25 START COUNTER() (1.1)

5.2.18 RECEIVE EXPRESSIONS

syntax:

<receive expression> ::= (1)
RECEIVE <buffer location> (1.1)

semantics: The receive expression delivers a value out of the specified buffer or from any delayed sending process. If the receive expression is executed while the buffer does not contain a value or no sending process is delayed on it, the executing process is delayed until a value is sent to the buffer (see chapter 8 for full details).

static properties: The class of the *receive expression* is the M-value class, where M is the buffer element mode of the mode of the *buffer location*.

dynamic conditions: The lifetime of the denoted buffer location must not end while the executing process is delayed on that buffer location.

examples:

16.49 RECEIVE USER_BUFFER (1.1)

5.2.19 *ZERO-ADIC OPERATOR*

syntax:

```
<zero-adic operator> ::=                                    (1)
     THIS                                                   (1.1)
```

semantics: The zero-adic operator delivers the unique instance value identifying the process executing it.

static properties: The class of the *zero-adic operator* is the *INSTANCE*-derived class.

5.3 *VALUES AND EXPRESSIONS*

5.3.1 *GENERAL*

syntax:

```
<value> ::=                                                 (1)
     <expression>                                           (1.1)
   | <undefined value>                                      (1.2)

<undefined value> ::=                                       (2)
     *                                                      (2.1)
   | <undefined synonym name>                               (2.2)
```

semantics: A value is either an undefined value or a (CHILL defined) value delivered as the result of the evaluation of an expression.

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.

dynamic properties: A value is said to be undefined if it is denoted by the *undefined value* or when explicitly indicated in this document. A composite value is undefined if and only if all its sub components (i.e. substring values, element values, field values) are undefined.

(Note: A *value* can denote an undefined value only in the following contexts:

- it is an *undefined value*;
- it is a *location contents*, containing an undefined value;
- it is a *value procedure call*, delivering an undefined value;
- it is a *value substring*, a *value string slice*, a *value array element*, a *value sub-array*, a *value array slice*, or a *value structure field*, delivering an undefined value.)

examples:

```
6.40    (146_097*c)/4+(1_461*y)/4
        +(153+m+c)/5+day+1_721_119                          (1.1)
```

5.3.2 EXPRESSIONS

syntax:

```
<expression> ::=                                            (1)
    <operand-1>                                             (1.1)
  | <sub expression> { OR | XOR } <operand-1>                (1.2)

<sub expression> ::=                                        (2)
    <expression>                                            (2.1)
```

semantics: 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 mixed order. They need only to be evaluated to the point that the value to be delivered is determined uniquely. If the expression is constant or literal, the evaluation will never cause an exception.

If *OR* or *XOR* is specified the *sub expression* and the *operand-1* deliver:

- boolean values, in which case *OR* and *XOR* denote the usual logical operators delivering a boolean value;
- bit string values, in which case *OR* and *XOR* denote the usual logical operations on 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 *expression* is an *operand-1*, the class of the *expression* is the class of the *operand-1*. If *OR* or *XOR* is specified, the class of the *expression* is the resulting class of the class of *sub expression* and the *operand-1*.

An *expression* is constant (literal) if and only if it is either an *operand-1* which is constant (literal), or built up from an *expression* and an *operand-1* which are both constant (literal).

static conditions: If *OR* or *XOR* is specified, the class of the *sub expression* must be compatible with the class of the *operand-1*. Both classes must have a boolean, powerset or bit string root mode.

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:

```
10.27   i<min                          (1.1)
10.27   i<min OR i>max                 (1.2)
```

5.3.3 *OPERAND-1*

syntax:

```
<operand-1> ::=                                   (1)
      <operand-2>                                 (1.1)
    | <sub operand-1> AND <operand-2>             (1.2)

<sub operand-1> ::=                               (2)
      <operand-1>                                 (2.1)
```

semantics: If *AND* is specified, *sub operand-1* and *operand-2* deliver:

- boolean values, in which case *AND* denotes the usual logical "and" operation, delivering a boolean value;
- bit string values, in which case *AND* denotes the usual logical "and" operation on 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 *operand-1* is an *operand-2*, the class of the *operand-1* is the class of the *operand-2*.

If *AND* is specified, the class of the *operand-1* is the resulting class of the classes of the *operand-2* and *sub operand-1*.

An *operand-1* is constant (literal) if and only if it is either an *operand-2* which is constant (literal), or built up from an *operand-1* and an *operand-2* which are both constant (literal).

static conditions: If *AND* is specified, the class of the *sub operand-1* must be compatible with the class of the *operand-2*. These classes must both have a boolean, powerset or bit string root mode.

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:

```
5.11    (a1 OR b1)                                  (1.1)
5.11    NOT k2 AND (a1 OR b1)                       (1.2)
```

5.3.4 *OPERAND-2*

syntax:

```
<operand-2> ::=                                     (1)
     <operand-3>                                    (1.1)
   | <sub operand-2> <operator-3> <operand-3>       (1.2)

<sub operand-2> ::=                                 (2)
     <operand-2>                                    (2.1)

<operator-3> ::=                                    (3)
     <relational operator>                          (3.1)
   | <membership operator>                          (3.2)
   | <powerset inclusion operator>                  (3.3)

<relational operator> ::=                           (4)
     = | /= | > | >= | < | <=                        (4.1)

<membership operator> ::=                           (5)
     IN                                             (5.1)

<powerset inclusion operator> ::=                   (6)
     <= | >= | < | >                                (6.1)
```

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 or string 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, testing whether or not a set value is contained in: <= , is properly contained in: <, contains: >=, or properly

contains: > the other set value. The powerset inclusion operator delivers a boolean value as result.

static properties: If an *operand-2* is an *operand-3*, the class of the *operand-2* is the class of the *operand-3*. If an *operator-3* is specified, the class of the *operand-3* 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 an *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 the *operand-3* must be fulfilled:

- if the *operator-3* is = or /=, both classes must be compatible;
- if the *operator-3* is a *relational operator* other than = or /=, both classes must be compatible and must have a discrete or string root mode;
- if the *operator-3* is a *membership operator*, the class of *operand-3* must have a powerset root mode and the class of the *sub operand-2* must be compatible with the member mode of that root mode;
- if the *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 parameterised structure mode.

examples:

```
10.46   NULL                                        (1.1)
10.46   last=NULL                                   (1.2)
```

5.3.5 *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 the operator-4 is an arithmetic additive operator, both operands deliver integer values and the resulting integer value is the sum (+) or difference (-) of the two values.

If the 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.

If the 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.

static properties: If an *operand-3* is an *operand-4*, the class of the *operand-3* is the class of *operand-4*. If an *operator-4* is specified, the class of the *operand-3* is determined by the *operator-4* as follows:

- if *operator-4* is a *string concatenation operator*, the class of the *operand-3* is, depending on the classes of the *operand-4* and *sub operand-3*:
  - if none of them is strong, the class is the *BIT(n)*-derived class or *CHAR(n)*-derived class, depending on whether both operands are bit or character strings, where *n* is the sum of the 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 mode of one of the strong operands and *n* denotes the sum of the length of the root modes of both classes

  (this class is dynamic if one or both operands have a dynamic class).

- if *operator-4* is an *arithmetic additive operator* or *powerset difference operator*, the class of the *operand-3* is the resulting class of the classes of the *operand-4* and the *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).

static conditions: If an *operator-4* is specified, the following compatibility requirements must be fulfilled:

- if *operator-4* is an *arithmetic additive operator*, the classes of both operands must be compatible and they must both have an integer root mode;

- if *operator-4* is a *string concatenation operator*, the root modes of the classes of both operands must both be bit string modes or both be character string modes and, if both classes are value classes, their root modes must have the same novelty;

- if *operator-4* is a *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* which is not constant , an *OVERFLOW* exception occurs if an addition (+) or a subtraction (-) gives rise to a value which is not within the bounds specified by the root mode of the class of the *operand-3*.

examples:

```
1.5     j                                                    (1.2)
1.5     i+j                                                  (1.2)
```

5.3.6 *OPERAND-4*

syntax

```
<operand-4> ::=                                              (1)
     <operand-5>                                             (1.1)
   | <sub operand-4>
     <arithmetic multiplicative operator> <operand-5>        (1.2)

<sub operand-4> ::=                                          (2)
     <operand-4>                                             (2.1)

<arithmetic multiplicative operator> ::=                     (3)
     * | / | MOD | REM                                       (3.1)
```

semantics: If an arithmetic multiplicative operator is specified, sub operand-4 and operand-5 deliver integer values and the resulting integer value is either the product (*), the quotient (/), modulo (*MOD*) or division remainder (*REM*) 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 $I = N * J + K$. *J* must be greater than *0*.

The remainder operation is defined such that *X REM Y = X - (X/Y) * Y* yields *TRUE* for all integer values *X* and *Y*.

static properties: If the *operand-4* is an *operand-5*, the class of the *operand-4* is the class of the *operand-5*, otherwise the class of the *operand-4* is the resulting class of the classes of the *sub operand-4* and the *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, the classes of the *operand-5* and *sub operand-4* must be compatible and both must have an integer root mode.

dynamic conditions: In the case of an *operand-4*, which is not constant, an *OVERFLOW* exception occurs if a multiplication (*) or a division (/) or a modulo (*MOD*) or a remainer (*REM*) operation gives rise to a value which is not in the set of values defined by the root mode of the class of the *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.

examples:

```
6.15    1_461                          (1.1)
6.15    (4 * d + 3) / 1_461            (1.2)
```

5.3.7 *OPERAND-5*

syntax

```
<operand-5> ::=                                   (1)
     [<monadic operator>] <operand-6>             (1.1)

<monadic operator> ::=                            (2)
     - | NOT                                      (2.1)
   | <string repetition operator>                 (2.2)
```

<string repetition operator> ::= *(3)*
(<integer literal expression>) *(3.1)*

semantics: If the monadic operator is a change-sign operator (-), the operand-6 delivers an integer value and the resulting integer value is the previous integer value with its sign changed.

If the monadic operator is *NOT*, the operand-6 delivers either a boolean value, or a bit string value or a powerset value. In the first two cases the logical negation of the boolean or 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.

If the monadic operator is a string repetition operator, the operand-6 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 string value formed by concatenating the string with itself as many times as specified by the value delivered by the literal expression minus 1.

static properties: If the *operand-5* is an *operand-6*, the class of the *operand-5* is the class of the *operand-6* .

If a *monadic operator* is specified, the class of the *operand-5* is:

- if the *monadic operator* is - or *NOT* then the resulting class of the *operand-6*;
- if the *monadic operator* is the *string repetition operator*, then it is the *CHAR(n)* or *BIT(n)*-derived class (depending on whether the literal was a *character string literal* or *bit string literal*) where $n = r * L$, where r is the value delivered by the *integer literal expression* and L is the length of the string literal.

An *operand-5* is constant (literal) if and only if the *operand-6* is constant (literal).

static conditions: If the *monadic operator* is -, the class of the *operand-6* must have an integer root mode.

If the *monadic operator* is *NOT*, the class of the *operand-6* must have a boolean, bit string or powerset root mode.

If the *monadic operator* is the *string repetition operator*, the *operand-6* must be a *character string literal* or a *bit string literal*. The *integer literal expression* must deliver a non-negative integer-value.

dynamic conditions: If the *operand-5* is not constant, an *OVERFLOW* exception occurs if a change sign (-) operation gives rise to a value which is not in the set of values defined by the root mode of the class of the *operand-5*.

examples:

5.11	*NOT k2*	*(1.1)*
7.50	*(6)' '*	*(1.1)*
7.50	*(6)*	*(2.2)*

5.3.8 OPERAND-6

syntax:

<operand-6> ::=	*(1)*
<primitive value>	*(1.1)*
| <parenthesised expression>	*(1.2)*
<parenthesised expression> ::=	*(2)*
(<expression>)	*(2.1)*

semantics: An operand-6 is either a primitive value (see section 5.2) or a parenthesised expression.

static properties: The class of the *operand-6* is the class of the *primitive value* or *parenthesised expression* respectively. The class of the *parenthesised expression* is the class of the *expression*.

An *operand-6* is constant (literal) if and only if the *primitive value* or *expression*, respectively is constant (literal).

examples:

1.5	*i*	*(1.1)*
5.11	*(a1 OR b1)*	*(1.2)*

6.0 ACTIONS

6.1 GENERAL

syntax:

```
<action statement> ::=                                              (1)
     [<name> :] <action> [<handler>] [<label name>];                 (1.1)

<action> ::=                                                        (2)
     <bracketed action>                                             (2.1)
   | <assignment action>                                            (2.2)
   | <call action>                                                  (2.3)
   | <exit action>                                                  (2.4)
   | <return action>                                                (2.5)
   | <result action>                                                (2.6)
   | <goto action>                                                  (2.7)
   | <assert action>                                                (2.8)
   | <empty action>                                                 (2.9)
   | <start action>                                                 (2.10)
   | <stop action>                                                  (2.11)
   | <delay action>                                                 (2.12)
   | <continue action>                                              (2.13)
   | <send action>                                                  (2.14)
   | <cause action>                                                 (2.15)

<bracketed action> ::=                                              (3)
     <if action>                                                    (3.1)
   | <case action>                                                  (3.2)
   | <do action>                                                    (3.3)
   | <module>                                                       (3.4)
   | <begin-end block>                                              (3.5)
   | <delay case action>                                            (3.6)
   | <receive case action>                                          (3.7)
```

semantics: Action statements constitute the algorithmic part of a CHILL program. Any action statement may be labelled and those actions that might cause an exception may have a handler appended.

static properties: A *name* followed by a colon and placed in front of an *action*, and only such a name, is defined to be a *label* name.

static conditions: The *label name* before the semicolon may only be given if the *action* is a *bracketed action* or if a *handler* is specified and only if a *name* followed by a colon is given before the *action*. The *label name* must be equal to the latter *name*.

6.2 ASSIGNMENT ACTION

syntax:

```
<assignment action> ::=                                              (1)
      <single assignment action>                                     (1.1)
    | <multiple assignment action>                                   (1.2)

<single assignment action> ::=                                       (2)
      <location> {<assignment symbol> | <assigning operator>}
      <value>                                                        (2.1)

<multiple assignment action> ::=                                     (3)
      <location> {,<location>}* <assignment symbol>
      <value>                                                        (3.1)

<assigning operator> ::=                                             (4)
      <closed dyadic operator> <assignment symbol>                   (4.1)

<closed dyadic operator> ::=                                         (5)
      OR | XOR | AND                                                 (5.1)
    | <powerset difference operator>                                 (5.2)
    | <arithmetic additive operator>                                 (5.3)
    | <arithmetic multiplicative operator>                           (5.4)

<assignment symbol> ::=                                              (6)
      := | =                                                         (6.1)
```

derived syntax: The = symbol is derived syntax for the := symbol.

semantics: The 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 the assignment themselves are performed in an unspecified and possibly mixed 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 variant fields that depend on it will receive an undefined value.

static conditions: The modes of all *location* occurrences must be equivalent and they must have neither the read-only property, nor the synchronisation 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.

If the *value* is a *regional expression* (see section 8.2.2), every *location* must be regional.

If in a *single assignment action* an *assigning operator* is specified, the specified *value* must be an *expression*.

dynamic conditions: The *TAGFAIL* exception occurs if, in the case of a dynamic parameterised structure mode location and/or value, the dynamic part of the above mentioned compatibility check fails.

The *RANGEFAIL* exception occurs if any *location* has range mode and the value delivered by the evaluation of *value* lies outside the bounds specified by that range mode.

The *RANGEFAIL* exception occurs if, in the case of a dynamic parameterised string mode or array mode location and/or value, the dynamic part of the above mentioned compatibility check fails.

The above mentioned conditions are called the assignment conditions of a value with respect to a mode (i.e. the mode of the location).

In the case of an *assignment 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).

examples:

```
4.11    a:=b+c                                  (1.1)
10.21   stackindex-:=1                          (2.1)
19.16   X.PREX, X.NEXT := NULL                  (3.1)
10.21   -:=                                     (4.1)
```

6.3 *IF ACTION*

syntax:

```
<if action> ::=                                 (1)
     IF <boolean expression> <then clause>
     [<else clause>] FI                         (1.1)
```

```
<then clause> ::=                                          (2)
     THEN <action statement list>                          (2.1)

<else clause> ::=                                          (3)
     ELSE <action statement list>                          (3.1)
   | ELSIF <boolean expression>
     <then clause> [<else clause>]                         (3.2)
```

derived syntax: The notation:

ELSIF <boolean expression> <then clause> [<else clause>]

is derived syntax for:

ELSE IF <boolean expression> <then clause> [<else clause>] FI;

semantics: The 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.

examples:

```
7.24    IF n >= 10 THEN rn(r):='X';
              n-:=10;
              r+:=1;
        FI                                                 (1.1)
10.46   IF last = NULL
           THEN first,last:=p;
           ELSE last->.succ:=p;
           p->.pred:=last;
           last:=p;
        FI                                                 (1.1)
```

6.4 CASE ACTION

syntax:

```
<case action> ::=                                          (1)
     CASE <case selector list> OF [<range list>;]
     {<case alternative>}*
     [ELSE <action statement list>]
     ESAC                                                  (1.1)

<case selector list> ::=                                   (2)
     <discrete expression> {,<discrete expression>}*       (2.1)

<range list> ::=                                           (3)
     <discrete mode> {,<discrete mode>}*                   (3.1)

<case alternative> ::=                                     (4)
     <case label specification> : <action statement list>  (4.1)
```

semantics: The 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 defines a set of values. The use of a list of discrete expresssions 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.

The expressions in the case selector list are evaluated in an undefined and possibly mixed order. They need to 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 section 9.1.3).

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* 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* 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 *discrete mode* in a *case label* (see section 9.1.3) must lie in the range of the corresponding *discrete mode* 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* of the *range list*, if present, must also lie in that range.

The optional keyword *ELSE*, followed by an *action statement list*, may only be omitted if the list of *case label list* occurrences is complete (see section 9.1.3).

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* in the *range list*.

examples:

```
4.10    CASE order OF
          (1): a:=b+c;
             RETURN;
          (2): d:=0;
          (ELSE): d:=1;
        ESAC                                              (1.1)
11.44   starting.p.kind, starting.p.color                 (2.1)
11.62   (rook),(*):
            IF NOT ok_rook(b,m)
              THEN
                CAUSE illegal;
            FI;                                           (4.1)
```

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: The do action has 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.

dynamic conditions: The *SPACEFAIL* exception occurs if the storage requirements cannot be satisfied.

examples:

```
4.16    DO FOR i:=1 TO c;
           op(a,b,d,order-1);
           d:=a;
        OD                                                (1.1)
15.48   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)
    <name>                                                (5.1)

<start value> ::=                                         (6)
    <expression>                                          (6.1)

<step value> ::=                                          (7)
    BY <integer expression>                               (7.1)

<end value> ::=                                           (8)
    TO <expression>                                       (8.1)

<range enumeration> ::=                                   (9)
    <loop counter> [DOWN] IN <discrete mode>              (9.1)

<powerset enumeration> ::=                                (10)
    <loop counter> [DOWN] IN <powerset expression>        (10.1)

<location enumeration> ::=                                (11)
    <loop counter> [DOWN] IN <array location>             (11.1)
```

semantics: The action statement list is repeatedly entered according to the specified for control.

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.

A distinction is made between normal and abnormal termination. Normal termination occurs if the evaluation of at least one of the loop counters indicates termination. Abnormal termination occurs if a while condition evaluation delivers *FALSE*, if an exit action or a goto action with a (target) label defined outside the action statement list is executed, or if an exception is caused for which the appropriate handler lies outside, and is not appended to, the do action.

1. do for ever:

   The action list is indefinitely repeated; only abnormal termination is possible.

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 (range enumeration), or by a powerset value (powerset enumeration), or by a start value, step value and end value (step enumeration).

   The loop counter is always implicitly defined inside the action statement list. However, if an access name which is equal to the name of the loop counter is visible outside the do action, the value of the loop counter will be stored into the denoted location just prior to abnormal termination. In the case of normal termination the value stored into the location denoted by the external access name is undefined.

   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. For subsequent executions of the action statement list, the *"next value"* will be evaluated as:
   SUCC("previous value") (*PRED("previous value")*).
   The do action is terminated (normal termination) if the action statement list has been executed for the greatest (smallest) value defined by the discrete mode.

   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, next value will be the next greater (smaller) member value in the powerset value. The do action is terminated (normal termination) when the action statement list has been executed for the greatest (smallest) value. When the do action is executed, the powerset expression is evaluated once only.

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, end value, and possibly step value. When the do action is executed, these expressions are evaluated once only in an unspecified, possibly mixed 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"))*.
The do action is terminated (normal termination) if the evaluation yields a value which is greater (smaller) than the end value, or would cause 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 specified locations which are the elements of the array location denoted by the *array location*. The semantics are as if initially the loc-identity declaration:
DCL <loop counter> <mode> LOC := <first location>;
were encountered, where *<mode>* is the element mode of the mode of the *array location* and *<first location>* the element of the smallest (greatest) index; for subsequent executions, as if before each execution of the action statement list the loc-identity declaration:
DCL <loop counter> <mode> LOC := <next location>;
where encountered, where *<next location>* is the array element with index:
"next index" = SUCC("previous index")
(PRED("previous index")).
The do action is terminated (normal termination) if the

loop counter just before the next evaluation indicates the array element with the greatest (smallest) index. When the do action is executed, the *array location* is evaluated once only.

static properties:

value enumeration:

The *loop counter* is a value enumeration name. If a name is visible in the reach in which the *do action* is placed which is equal to the *loop counter*, the *loop counter* is explicit, otherwise it is implicit.

step enumeration:

The class of an explicit *loop counter* is the M-value class, where M is the mode of the external access name (see below: static conditions).

The class of an implicit *loop counter* is the resulting class of the classes of the *start value*, *step value* if present, and *end value*.

range enumeration:

The class of the *loop counter* is the M-value class, where M is the *discrete mode*.

powerset enumeration:

The class of 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 *loop counter* is a location enumeration name. Its mode is the element mode of the mode of the *array location*.

A location enumeration name is (language) referable if the element layout of the mode of the *array location* is *NOPACK*.

static conditions:

step enumeration:

The classes of *start value*, *end value* and *step value*, if present, must be pairwise compatible. In the case of a *loop counter* which is explicit, the externally visible name must be an access name. The mode of the external access name must be compatible with each of these classes and must not be a read-only mode.

powerset enumeration, range enumeration:

In the case of an explicit *loop counter*, the externally visible name must be an access name. The mode of the external access name must be compatible with the class of the *loop counter*.

The *powerset expression* must be strong.

dynamic conditions: A *RANGEFAIL* exception occurs if the value delivered by *step value* is not greater than 0 or if, in the case of an explicit *loop counter*, the value to be stored back into the external location prior to abnormal termination, does not lie within the bounds specified by the mode of the external location. This exception occurs outside the block of the do action.

examples:

```
4.16    FOR i:=1 TO c                                  (1.1)
15.27   FOR EVER                                       (1.1)
4.16    i:=1 TO c                                      (3.1)
9.11    j:=MIN(sieve) BY MIN(sieve) TO max             (3.1)
14.22   I IN INT(1:100)                                (3.2)
```

6.5.3 *WHILE CONTROL*

syntax:

```
<while control> ::=                                    (1)
     WHILE <boolean expression>                        (1.1)
```

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 *TRUE*, the action statement list is entered, otherwise the do action is terminated (abnormal termination).

examples:

```
7.28    WHILE n >= 1                                   (1.1)
```

6.5.4 *WITH PART*

syntax:

```
<with part> ::=                                        (1)
     WITH <with control> {,<with control>}*            (1.1)

<with control> ::=                                     (2)
     <structure location>                              (2.1)
   | <structure expression>                            (2.2)
```

N.B. if the *structure expression* is a *location*, the syntactic construct is ambiguous and will be interpreted as a *structure location*.

semantics: The (visible) field names of the structure locations or structure value specified in each *with control* are made available as direct accesses to the fields.

If a *structure location* is specified, access names which are equal to the field names of the mode of the *structure location* are implicitly created, denoting the sub-locations of the structure location.

If a *structure expression* is specified, value names which are equal to the field names of the mode of the (strong) *structure expression* are implicitly created, 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 an unspecified, possibly mixed order.

static properties:

Structure expression: Any name made available in the do action is 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 expression*, which is made available as value do-with name.

Structure location: Any name made available in the do action is 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 (language) referable if the field layout of the associated field name is *NOPACK*.

static conditions: The *structure expression* must be strong.

examples:

15.48 WITH EACH *(1.1)*

6.6 *EXIT ACTION*

syntax:

<exit action> ::= *(1)*
EXIT <label name> *(1.1)*

semantics: An exit action is used to leave a bracketed action. Action is resumed immediately after the closest surrounding bracketed action labelled with the label name.

static conditions: The *exit action* must lie within the bracketed action statement labelled with the *label name*. If the *exit action* is placed within a procedure or process definition, the exited bracketed action statement 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:

```
15.52    EXIT FIND_COUNTER                                    (1.1)
```

6.7 CALL ACTION

syntax:

```
<call action> ::=                                             (1)
     [CALL] { <procedure call> | <built-in routine call>}     (1.1)

<procedure call> ::=                                          (2)
     {< procedure name> | <procedure expression>}
     ([<actual parameter list>])                              (2.1)

<actual parameter list> ::=                                   (3)
     <actual parameter> {,<actual parameter>}*                (3.1)

<actual parameter> ::=                                        (4)
     <value>                                                  (4.1)
   | <static mode location>                                   (4.2)
```

derived syntax: The keyword *CALL* is optional. A *call action* with *CALL* is derived from a *call action* without *CALL*.

semantics: A call action causes a call of the general procedure indicated by the value delivered by the procedure expression or the procedure indicated by the procedure name. The actual values and locations specified in the actual parameter list are passed to the procedure.

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 may be regional (the latter is only possible with a *procedure name*, see section 8.2.2). These properties are inherited from the *procedure name* or any mode compatible with the class of the *procedure expression* (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*.

static conditions: 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 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 synchronisation property. If the *procedure call* is not regional, the (actual) *value* must not be regional (see section 8.2.2).

- If the parameter spec has the *INOUT* or *OUT* attribute, the *actual parameter* must be a *static mode 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) *static mode location* must not have the read-only property nor the synchronisation property. If the *procedure call* is not regional, the (actual) *location* must not be regional (see section 8.2.2).

- 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 *static mode location*.

- If the parameter spec has the *LOC* attribute, the *actual parameter* must be a *static mode location* which is both referable and such that the mode in the parameter spec is read-compatible with the mode of this (actual) *static mode location*, or a *value* which is not a *location* but whose class is compatible with the mode in the parameter spec.

dynamic conditions: A procedure call can cause any of the exceptions of the attached set of exception names. It causes the *EMPTY* exception if the *procedure expression* delivers *NULL*, it causes the *SPACEFAIL* exception if storage requirements cannot be satisfied and it causes the *RECURSEFAIL* exception if the procedure calls itself recursively (i.e. a previous invocation is still active) and its recursivity is non-recursive.

Parameter passing can cause the following exceptions:

- If the parameter spec has the *IN*, *INOUT* or *LOC* attribute, the assignment conditions of the (actual) value (possibly contained in an actual location), with respect to the mode of the parameter spec apply at the point of the call (see section 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 section 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 section 6.2).

The *procedure expression* 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 (see section 8.1) and the lifetime of the denoted procedure must not have ended.

examples:

```
4.17    op(a,b,d,order-1)                                   (1.1)
```

6.8 RESULT AND RETURN ACTION

syntax:

```
<return action> ::=                                         (1)
     RETURN [<result>]                                      (1.1)

<result action> ::=                                         (2)
     RESULT <result>                                        (2.1)

<result> ::=                                                (3)
     <value>                                                (3.1)
   | <static mode location>                                 (3.2)
```

derived syntax: The *return action* with *result* is derived from *RESULT <result> ; RETURN*. If a *handler* is appended to such a *return action*, it is considered to be appended to the *result action* from which it was derived.

semantics: The result action serves to establish the result to be delivered by a procedure call. This result may be a location or a value. The 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 last executed result action. If no result action has been executed the procedure call delivers an undefined location or undefined value, respectively.

static properties: The *result action* and *return action* have a procedure name attached, which is the name of the closest surrounding procedure definition.

static conditions: The *return action* and the *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* is specified in the result spec of the procedure name of the *result action*, the *result* must be a *static mode location*, such that the mode in the result spec is read-compatible with the mode of the *static mode location*. If the procedure name of a *result action* is not regional, the *static mode location* in the *result* must not be regional (see section 8.2.2).

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. If the procedure name of a *result action* is not regional, the *value* in the *result* must not be regional (see section 8.2.2).

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:

```
4.20    RETURN                                  (1.1)
1.5     RESULT i+j                              (2.1)
5.20    c                                       (3.1)
```

6.9 GOTO ACTION

syntax:

```
<goto action> ::=                               (1)
    GOTO <label name>                           (1.1)
```

semantics: The goto action causes a transfer of control. Action is resumed with the action statement labelled with the label name.

static conditions: If the *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: The assert action provides a means of testing a condition.

dynamic conditions: The *ASSERTFAIL* exception occurs if the *boolean expression* delivers *FALSE*.

examples:

```
4.6      ASSERT b>0 AND c>0 AND order>0                (1.1)
```

6.11 EMPTY ACTION

syntax:

```
<empty action> ::=                                     (1)
     <empty>                                           (1.1)

<empty> ::=                                            (2)
```

semantics: The empty action does not cause any 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: The cause action causes an exception.

static conditions: A *handler* must not be appended to a *cause action*.

dynamic conditions: The *cause action* causes the exception whose name is indicated by *exception name*.

examples:

```
4.8      CAUSE wrong_input                             (1.1)
```

6.13 START ACTION

syntax:

<start action> ::= (1)
<start expression> [SET <instance location>] (1.1)

derived syntax: The *start action* with the *SET* option is derived syntax for the single assignment action:
<instance location> := <start expression>

semantics: The start action evaluates the start expression (see section 5.2.17), without using the resulting instance value.

examples:

14.37 START CALL_DISTRIBUTOR() (1.1)

6.14 STOP ACTION

syntax:

<stop action> ::= (1)
STOP (1.1)

semantics: The stop action terminates the process executing the stop action (see section 8.1).

static conditions: A *handler* must not be appended to a *stop action*.

6.15 CONTINUE ACTION

syntax:

<continue action> ::= (1)
CONTINUE <event location> (1.1)

semantics: The continue action allows the process of the highest priority, which is delayed on the specified event location, to be activated. If there is no unique process of the highest priority, one particular process of the highest possible priority will be selected according to an implementation defined scheduling algorithm. If there are no processes delayed on the specified event location, the continue action has no further effect (see chapter 8 for further details).

examples:

13.23 CONTINUE RESOURCE_FREED (1.1)

6.16 DELAY ACTION

syntax:

```
<delay action> ::=                                              (1)
    DELAY <event location> [<priority>]                         (1.1)

<priority> ::=                                                  (2)
    PRIORITY <integer literal expression>                       (2.1)
```

semantics: The delay action causes the process executing it to become delayed. It can become activated by a continue action on the event location specified. The priority indicates the priority of the delayed process within the set of processes which are delayed on the indicated event location. The default and lowest priority is 0 (see chapter 8 for further details).

static conditions: The *integer literal expression* must not deliver a negative value.

dynamic conditions: The *DELAYFAIL* exception occurs if the mode of the *event location* has a length attached and the number of processes delayed on the specified event location is equal to the length just after the evaluation of the event location. This exception occurs before the delaying of the process.

The lifetime of the delivered event location must not end while the process executing the delay action is delayed on it.

examples:

```
13.17   DELAY RESOURCE_FREED                                    (1.1)
```

6.17 DELAY CASE ACTION

syntax:

```
<delay case action> ::=                                         (1)
    DELAY CASE [SET <instance location>;] [<priority>;]
    {<delay alternative>}*
    ESAC                                                        (1.1)

<delay alternative> ::=                                         (2)
    (<event list>) : <action statement list>                    (2.1)

<event list> ::=                                                (3)
    <event location> {,<event location>}*                       (3.1)
```

semantics: The delay case action causes the process executing it to become delayed. It can become activated by a continue action on one of the specified event locations. In that case an action statement list that is labelled by the event location on which the continue action, that re-activated the process,

was performed, will be executed (see chapter 8 for further details). Before the process becomes delayed, each event location and the instance location if specified, will be evaluated. They will all be evaluated in an unspecified and possibly mixed order. If two or more evaluations deliver the same event location, the choice of an action statement list is non-deterministic.

If an instance location is specified, the instance value identifying the process that executed the activating continue action, will be stored into the instance location.

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 the mode of at least one *event location* has a length attached such that the number of delayed processes on the specified event location is equal to the length after the evaluation of the *event location*. This exception occurs before the delaying of the process.

The lifetime of none of the delivered event locations must end while the process executing the delay case action is delayed on it.

examples:

```
14.20   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                                                    (1.1)
```

6.18 SEND ACTION

6.18.1 GENERAL

syntax:

```
<send action> ::=                                               (1)
      <send signal action>                                      (1.1)
    | <send buffer action>                                      (1.2)
```

semantics: The send action initiates the transfer of synchronisation information, from a sending process. The detailed semantics depend on whether the synchronisation 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 expression>] [<priority>] (1.1)

semantics: The specified signal is sent together with the list of values and priority (if present). The default and lowest priority is 0. If the signal name has a process name attached, it means that only processes of that name may receive the signal. If the *TO* option is specified, it identifies the only process that may receive the list of values sent in the send signal action. This process identification must not be in contradiction with a possible process name attached to the signal name. Both the possible process name of the signal and the possible instance value are dynamically attached to the list of values sent (see chapter 8 for further details).

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 regional (see section 8.2.2). 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 expression* delivers *NULL*.

The *EXTINCT* exception occurs if and only if the lifetime of the process indicated by the value delivered by the *instance expression* has terminated at the point of the execution of the send signal action.

The *MODEFAIL* 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 expression*.

examples:

```
15.68   SEND READY TO RECEIVED_USER                    (1.1)
15.76   SEND READOUT(COUNT) TO USER
```

6.18.3 SEND BUFFER ACTION

syntax:

<send buffer action> ::= (1)
SEND <buffer location>(<value>) [<priority>] (1.1)

semantics: The specified value together with the priority is stored into the buffer location if its capacity allows for it. The latter is not the case if the mode of the *buffer location* has a length attached and the number of values stored in the buffer is equal to the length just prior to the execution of the send buffer action. As a result, the sending process will become delayed until there is capacity in the buffer location or until the value sent is consumed. The default and lowest priority is 0 (see chapter 8 for further details).

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 regional (see section 8.2.2). The *integer literal expression* in *priority* must not deliver a negative value.

dynamic conditions: For the *send buffer action* 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 delaying of the process.

The lifetime of the delivered buffer location must not end while the process executing the send buffer action is delayed on it.

examples:

```
16.115  SEND USER->([READY, ->COUNTER_BUFFER])                    (1.1)
```

6.19 *RECEIVE CASE ACTION*

6.19.1 *GENERAL*

syntax:

```
<receive case action> ::=                                         (1)
      <receive signal case action>                                (1.1)
    | <receive buffer case action>                                (1.2)
```

semantics: The receive case action receives synchronisation information that is transmitted by the send action. The detailed semantics depend on the synchronisation 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 process (see chapter 8 for further details).

6.19.2 *RECEIVE SIGNAL CASE ACTION*

syntax:

<receive signal case action> ::= (1)
RECEIVE CASE [SET <instance location>;]
{<signal receive alternative>}+
[ELSE <action statement list>] ESAC (1.1)

<signal receive alternative> ::= (2)
(<signal name> [IN <name list>])
: <action statement list> (2.1)

semantics: The receive signal case action receives a signal, possibly with a list of values, the signal name of which is specified in a signal receive alternative.

When the receive signal case action is entered, and if a signal of one of the specified names which may be received by a process executing it is present for reception, the signal is received. If no such signal is present and if *ELSE* is not specified, the process executing the receive signal case action becomes delayed; if *ELSE* is specified, the action statement list following it will be entered.

A signal may be received by a process only if the following conditions are fulfilled:

- If a process name is attached to the signal, the name of the receiving process is that process name.
- If an instance value is attached to the signal it identifies the receiving process.

If a signal may be received, the action statement list labelled with the signal name of the received signal, will be entered. If more than one signal may be received, a signal of the highest priority will be selected according to an implementation defined scheduling algorithm. If the signal name has a list of modes attached, i.e. a list of values is sent with the signal, a list of names must be specified after *IN*. They are introduced value names denoting the received values. If in the reach in which the receive signal case action is placed, an access name is visible which is equal to an introduced name, the received value will be stored into the denoted location immediately after signal reception and before the execution of the action statement list.

If the *SET* option is specified, the instance value denoting the process that has sent the received signal, will be stored into the specified instance location immediately after signal reception.

static properties: Any name defined in the *name list* of the *signal receive alternative* is a value receive name. Its class is the M-value class, where M is the corresponding mode of the *signal name* in front of it. If a name is visible in the reach where the *signal receive case action* is placed, which is equal to one of the names introduced after *IN*, the value receive name is explicit, otherwise it is implicit.

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 *name 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 *name list* must be equal to the number of modes of the *signal name*.

If the value receive name is explicit, the externally visible name must be an *access name* and its mode must be compatible with the class of the value receive name. The mode of the *access name* must not have the read-only property.

dynamic conditions: If the value receive name is explicit the assignment conditions of the received value with respect to the mode of the external *access name* apply. The possible execptions occur after receiving the signal and before entering the action statement list.

The *SPACEFAIL* exception occurs if, when entering an action statement list, storage requirements cannot be satisfied.

examples:

```
15.73   RECEIVE CASE
          (STEP): COUNT +:= 1;
          (TERMINATE):
            SEND READOUT(COUNT) TO USER;
            EXIT WORK_LOOP;
        ESAC                                                    (1.1)
```

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                                                       (1.1)

<buffer receive alternative> ::=                                (2)
     (<buffer location> IN <name>)
```

: <action statement list> *(2.1)*

semantics: The receive buffer case action receives a value from a buffer location or from a sending process delayed on a buffer location, which location is indicated in a buffer receive alternative.

When the receive buffer case action is entered and if a value is present in, or a sending process is delayed on, one of the specified buffer locations, the value will be received and an action statement list labelled with a *buffer location* delivering the buffer location from which the value has been received, will be executed.

When the receive buffer case action is entered, the buffer locations are evaluated in an unspecified and possobly mixed order and they need only be evaluated up to a point sufficient to select an alternative. If none of the specified buffer locations contains a value and no sending process is delayed on a specified buffer location then if *ELSE* is not specified the executing process becomes delayed, if *ELSE* is specified the action statement list following it will be executed. If more than one value can be received, a value with the highest priority will be selected according to an implementation defined scheduling algorithm. If two or more *buffer location* occurrences deliver the same buffer location from which the value is received, the selection of the action statement list is non-deterministic.

The value is received immediately before entering the action statement list following the colon. The name after *IN* is an introduced value receive name denoting the received value. If in the reach where the *buffer receive case action* is placed, an access name is visible which is equal to a created value receive name, the received value is stored into the denoted location immediately before entering the action statement list.

If the *SET* option is specified, the specified instance location has stored in it, immediately on reception, the instance value denoting the process that has sent the received value.

static properties: The *name* after *IN* in the *buffer receive alternative* is 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*.

If a name is visible in the reach where the *receive buffer case action* is placed, which is equal to the *name* introduced after *IN*, the value receive name is called explicit, otherwise it is implicit.

static conditions: The mode of the *instance location* must not have the read-only property. If the value receive name is explicit, the externally visible name must be an *access name* and its mode must be compatible with the class of the value receive name with the same name. This mode must not have the read-only property.

dynamic conditions: If the value receive name is explicit the assignment conditions of the received value with respect to the mode of the external *access name* apply. The possible exceptions occur after receiving the value and before entering the action statement list.

The *SPACEFAIL* exception occurs if, when entering an action statement list, storage requirements cannot be satisfied.

The lifetime of none of the delivered buffer locations must end while the process executing the receive buffer case action is delayed on it.

7.0 PROGRAM STRUCTURE

7.1 GENERAL

The bracketed *do action*, *begin-end block*, *module*, *region*, *delay case action*, *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 will be used to denote:
  - the *action statement list* in the *do action* including the *loop counter* and *while control*;
  - the *begin-end block*;
  - the *procedure definition* excluding the *result spec*;
  - the *process definition*;
  - the *action statement list* in a *buffer receive alternative* or in a *signal receive alternative* including the *name* or *name list* after *IN*;
  - the *action statement list* after *ELSE* in a *receive case action* or *handler*;
  - the *on-alternative* in a *handler*.
- The word modulion will be used to denote either a *module* or a *region*.
- The word group will denote either a block or a modulion.
- The word reach or reach of a group will denote that part of the group which is not surrounded by an inner group of the group (i.e. the part consisting of the outermost nesting level of the group).

A group defines a scope for names created in its reach. Names can be created in the following ways:

- A name appearing in the *name list* of a *declaration*, *mode definition* or *synonym definition* or appearing in a *signal definition* is created in the reach where the *declaration*, *mode definition*, *synonym definition* or *signal definition*, respectively, is placed.
- A name appearing in the *name list* in a *formal parameter list* is created in the reach of the associated *procedure definition* or *process definition*.

- A name in front of a colon followed by an *action, region, procedure definition, entry definition* or *process definition* is created in the reach where the *action, region, procedure definition, procedure definition* containing the *entry definition, process definition,* respectively, is placed.

- Each value enumeration name, location enumeration name, value do-with name and location do-with name is created in the reach of the block of the associated *do action*.

- Each value-receive name is created in the reach of the block of the associated *signal receive alternative* or *buffer receive alternative*.

- A field name or set element name is created in the reach where the defining occurrence of its associated *structure mode* or *set mode* is placed.

- An exception name is created by means of a *cause action* or *on-alternative* (note: no specific point of creation is given for an exception name; see chapter 10).

- A language pre-defined name is considered to be created in the reach of a standard prelude module (see section 7.8).

Programmer introduced (created) names, except exception names, have a unique place where they are created (declared or defined). 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 (see section 9.2.8). No distinction between defining and applied occurrences is made for exception names (see chapter 10).

A name has a certain scope, i.e. that part of the program where its definition or declaration can be seen and, as a consequence, where it may be freely used. The name is said to be visible in that part. Locations 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. Modulions determine only visibility; the lifetime of locations created in the reach of a modulion will be the same as if they were created in the reach of the first surrounding block. Modulions 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.

7.2 REACHES AND NESTING

syntax:

```
<begin-end body> ::=                                              (1)
    <data statement list> <action statement list>                 (1.1)

<proc body> ::=                                                   (2)
```

```
        <data statement list>
        {<action statement> | <entry statement>}*                (2.1)

    <process body> ::=                                           (3)
        <data statement list> <action statement list>            (3.1)

    <module body> ::=                                            (4)
        {<data statement> | <visibility statement> |
         <region> }* <action statement list>                     (4.1)

    <region body> ::=                                            (5)
        {<data statement> | <visibility statement>}*             (5.1)

    <action statement list> ::=                                  (6)
        {<action statement>}*                                    (6.1)

    <data statement list> ::=                                    (7)
        {<data statement>}*                                      (7.1)

    <data statement> ::=                                         (8)
        <declaration statement>                                  (8.1)
      | <definition statement>                                   (8.2)

    <definition statement> ::=                                   (9)
        <synmode definition statement>                           (9.1)
      | <newmode definition statement>                           (9.2)
      | <synonym definition statement>                           (9.3)
      | <procedure definition statement>                         (9.4)
      | <process definition statement>                           (9.5)
      | <signal definition statement>                            (9.6)
      | <empty>;                                                 (9.7)
```

semantics: When a reach of a block is entered, all the lifetime-bound initialisations of the locations created when entering the block, are performed. Subsequently the reach-bound initialisations in the block reach and the possibly dynamic evaluations in the loc-identity declarations are performed in the order they are textually specified.

When a reach of a modulion is entered, the reach-bound initialisations and the possibly dynamic evaluations in the loc-identity declarations in the modulion reach are performed in the order they are textually specified.

static properties: Any reach has a unique directly enclosing group defined as follows:

- If the reach is the reach of a *do action*, *begin-end block*, *procedure definition*, *process definition*, *module* or *region*, then its directly enclosing group is the group in whose reach the *do action*, *begin-end block*, *procedure definition*, *process definition*, *module* or *region*, respectively, is placed.

- If the reach is the *action statement list*, possibly including introduced names, of 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 its directly enclosing group is the group in whose reach the *receive buffer case action* or *receive signal case action* is placed.

- 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 the directly enclosing group is the group in whose reach the statement, to which the *handler* is appended, is placed.

- If the reach is an *on-alternative* or *action statement list* after *ELSE*, of a *handler* which is appended to a group, then its directly enclosing group is the group to which the *handler* is appended.

A reach has a unique directly enclosing reach, which is the reach of the directly enclosing group. A statement has a unique directly enclosing group, which is the group in which reach the statement is placed. A reach is said to directly enclose a group (reach) if and only if the reach is the 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 the 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 its main entry (i.e. not via an additionally defined entry point).

- Process reach: the process is activated via a start statement.

- 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.

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.

7.3 BEGIN-END BLOCKS

syntax:

```
<begin-end block> ::=                                   (1)
    BEGIN <begin-end body> END                          (1.1)
```

semantics: A begin-end block is an action (compound action), possibly containing local declarations and definitions. It determines both visibility of locally created names and the lifetime of locally created locations (see sections 7.9 and 9.2.5).

dynamic conditions: A *SPACEFAIL* exception occurs if the *begin-end block* requires local storage for which storage requirements cannot be satisfied.

examples:

see 15.63 - 15.80

7.4 PROCEDURE DEFINITIONS

syntax:

```
<procedure definition statement> ::=                    (1)
    <name> : <procedure definition>
    [<handler>] [procedure name>];                      (1.1)

<procedure definition> ::=                              (2)
    PROC ([<formal parameter list>]) [<result spec>]
    [EXCEPTIONS(<exception list>)] <procedure attributes>;
    <proc body> END                                     (2.1)

<formal parameter list> ::=                             (3)
    <formal parameter> {,<formal parameter>}*           (3.1)
```

```
<formal parameter> ::=                                   (4)
     <name list> <parameter spec>                        (4.1)

<procedure attributes> ::=                               (5)
     [<generality>] [RECURSIVE]                          (5.1)

<generality> ::=                                         (6)
     GENERAL                                             (6.1)
   | SIMPLE                                              (6.2)
   | INLINE                                              (6.3)

<entry statement> ::=                                    (7)
     <name> : <entry definition>;                        (7.1)

<entry definition> ::=                                   (8)
     ENTRY                                               (8.1)
```

derived syntax: A *formal parameter*, where *name list* consists of more than one name, is derived from several *formal parameter* occurrences, separated by commas, one for each name 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 defines a (possibly) parameterised sequence of actions that may be called from different places in the program. Control is returned to the calling point either by executing a return action, or when reaching the end of the proc-body or an on-alternative of a handler appended to the procedure definition (falling through). Different degrees of complexity of procedures may be specified as follows:

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.

General procedures (*GENERAL*) do not have the restrictions of simple procedures and may be treated as procedure values.

Inline procedures (*INLINE*) have the same restrictions as simple procedures and they cannot be recursive. They have the same semantics as normal procedures, but the compiler will insert the generated object code at the point of invocation rather than generating code for actually calling the procedure.

Only simple and general procedures may be specified to be (mutually) recursive. When no procedure attributes are specified, an implementation default will apply.

A procedure may return a value or it may return a location (indicated by the *LOC* attribute in the result spec).

The name in front of the procedure definition defines the name of the procedure. If the procedure name is general, it is a procedure literal for the defined procedure value. Its class is determined by the modes and attributes in the formal parameter list and result spec.

A procedure may have multiple entry points by means of entry statements. These statements are considered as additional procedure definitions. The name in the entry statement defines the name of the entry point in the procedure in which reach it is placed. The entry point is determined by the textual position of the entry statement.

parameter passing:

There are basically two parameter passing mechanisms: the pass by value and the pass by location (*LOC* attribute). The attributes *OUT* and *INOUT* indicate variations of the pass by value mechanism.

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 <formal parameter name><mode> := <actual parameter>;
were encountered. However, the initialisation cannot cause an exception inside the procedure body. 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 the pass by location parameter passing, a location is passed as a parameter to the procedure body. Neither non-referable locations, nor dynamic mode 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 <formal parameter name><mode> LOC := <actual parameter>;

were encountered. However, such a declaration cannot cause an exception inside the procedure body.

If a *value* is specified which is not a *static mode 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.

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 *static mode location* (see section 6.8). 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 section 4.2.10) nor as a value procedure call (see section 5.2.15), but only as a call action (section 6.7).

register specification:

Register specification can be given in the formal parameter of the procedure, and in the result spec. In the pass by value case, it means that the actual value is contained in the specified register; in the pass by location case, it means that the (hidden) pointer to the actual location is contained in the specified register. If it is specified in the result spec it means that the returned value or the (hidden) pointer to the returned location is contained in the specified register.

static properties: A name is a procedure name if and only if it is defined in a *procedure definition statement* or in an *entry statement* (i.e. placed in front of a colon and a *procedure definition* or *entry definition*).

A procedure name has a *procedure definition* attached which is defined as:

- If the procedure name is defined in a *procedure definition statement* then the *procedure definition* in that statement.
- If the procedure name is defined in an *entry statement*, then the *procedure definition* in whose reach the *entry statement* is placed.

A procedure name has the following properties attached, defined by its *procedure definition*:

- It has a list of parameter specs, which are defined by the *parameter spec* occurrences in the *formal parameter list*, each parameter consisting of a mode, possibly a parameter attribute and/or register name.

- It has possibly a result spec, consisting of a mode, possibly a *LOC* attribute and/or register name.

- It has a possibly empty set of exception names, which are the names mentioned in *exception list*.

- It has a generality, which is, if *generality* is specified then 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 region, its generality is simple.

- It has a recursivity which is recursive if *RECURSIVE* is specified, otherwise an implementation default specifies either recursive or non-recursive. However, if the generality is inline, or if the procedure name is critical (see section 8.2) the recursivity is non-recursive.

A procedure name which is general, is a procedure literal. A general procedure name has a procedure mode attached, which is formed as:
PROC([<parameter list>]) [<result spec>]
[EXCEPTIONS(<exception list>)] [RECURSIVE]
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 comma's.

A name defined in a *name 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 (language) referable.

static conditions: If a procedure name is regional (see section 8.2.2), its procedure definition must not specify *GENERAL*.

If a procedure name is critical (see section 8.2), its definition may neither specify *GENERAL*, nor *RECURSIVE*.

No procedure definition may specify both *INLINE* and *RECURSIVE*.

If specified, the optional *procedure name* before the semicolon must be equal to the *name* 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 synchronisation property.

examples:

```
1.3     add:
        PROC(i,j INT) (INT) EXCEPTIONS(OVERFLOW);
           RESULT i+j;
        END add;                                          (1.1)
```

7.5 PROCESS DEFINITIONS

syntax:

```
<process definition statement> ::=                        (1)
     <name> : <process definition>
     [<handler>] [<process name>];                        (1.1)

<process definition> ::=                                  (2)
     PROCESS ([<formal parameter list>]);
     <process body> END                                   (2.1)
```

semantics: A process definition defines a possibly parameterised sequence of actions that may be started for concurrent execution from different places in the program (see chapter 8).

static properties: A name is a process name if and only if it is defined in a *process definition statement* (i.e. placed in front of a colon and a *process definition*).

A process name may have an implementation defined set of exception names attached.

static conditions: If specified, the optional *process name* before the semicolon must equal to the *name* in front of the *process definition*.

A *process definition statement* must not be surrounded by a region, nor by a block other than the imaginary outermost process definition (see section 7.8).

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 synchronisation property.

examples:

```
14.12   PROCESS();
          DO FOR EVER;
            WAIT(10 /* seconds */);
```

```
        CONTINUE OPERATOR_IS_READY;
    OD;
END                                                    (2.1)
```

7.6 MODULES

syntax:

```
<module> ::=                                           (1)
    MODULE <module body> END                           (1.1)
```

semantics: A module is an action possibly containing local declarations and definitions. A module is a means of restricting the visibility of names; it does not influence the lifetime of the locally created locations.

The detailed visibility rules for modules are given in section 9.2.

static properties: A name is a module name if and only if it is defined by placing it in front of a colon before *MODULE*.

examples:

```
7.42    MODULE
          SEIZE convert;
          DCL n INT INIT := 1979;
          DCL rn CHAR(20) INIT :=(20)' ';
          GRANT n, rn;
          convert();
          ASSERT rn = 'MDCCCCLXXVIIII'//(6)' ';
        END                                            (1.1)
```

7.7 REGIONS

syntax:

```
<region> ::=                                           (1)
    [<name> :] REGION <region body> END
    [<handler>] [<region name>];                       (1.1)
```

semantics: A region is a means of providing mutually exclusive access to its locally declared data object for the concurrent executions of processes (see chapter 8). It determines visibility of locally created names in the same way as a module.

static properties: A name is a region name if and only if it is defined by placing it in front of a colon before *REGION*.

static conditions: The optional *region name* before the semicolon must be equal to the *name* of the region.

A *region* must not be surrounded by a block other than the imaginary outermost process definition.

examples:

see 13.1 - 13.25

7.8 PROGRAM

syntax:

```
<program> ::=                                                    (1)
    {<module action statement> | <region>}+                      (1.1)
```

semantics: Programs consist of a list of modules or regions, surrounded by an imaginary outermost process definition. This process definition is considered to contain in its reach a standard CHILL prelude module. This module contains the definitions of the CHILL pre-defined names and the implementation pre-defined built-in routines, modes and register names.

static properties: The language and implementation defined names (see Appendix C2) are considered to be created in a module in the reach of the imaginary outermost process definition and granted *PERVASIVE* by that module (see section 9.2.6.2).

7.9 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* 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, 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 the *GETSTACK* built-in routine call is the time between that execution and the leaving of the closest surrounding block. If the *GETSTACK* built-in routine call is executed while evaluating an actual parameter of a procedure call or start expression, the lifetime of the created location will be the procedure call or the lifetime of the created process.

The lifetime of an access created in a loc-identity declaration is the closest surrounding block of the loc-identity declaration.

The lifetime of a procedure is the closest surrounding 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* which is declared with the attribute *STATIC*, or whose definition is not surrounded by a block other than the imaginary process definition.
- A *loc-identity name* such that the *static mode location* occurring in its definition is static.
- A *string element* or *substring* where the *string location* is static and either the *left element* and *right element*, or *position* are constant.
- An *array element* or *sub-array* where the *array location* is static and either the *expression*, or the *lower element* and the *upper element*, or the *integer expression* occurring in it are constant.
- A *structure field* where the *structure location* is static. If the *structure location* is not a *parameterised structure location* then the *field name* must not be a *variant field name*.
- A *location conversion* where the *location* occurring in it is static.

8.0 CONCURRENT EXECUTION

8.1 PROCESSES AND THEIR DEFINITIONS

A process is the sequential execution of a series of statements, the sequential execution of which may be concurrent with other processes. The behaviour of a process is described by a process definition (see section 7.5), which 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 section 5.2.17). It becomes active (i.e. under execution) and is considered to be executed concurrently with other processes. 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 section 7.9), 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 section 7.8), 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 or by reaching the end of the process body or the end of an on-alternative of 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 its subsidiary processes (i.e. processes created by start expressions in it) are terminated.

A process is, at the CHILL programming level, always in one of two states: it is either active (i.e. under execution) or delayed (i.e. waiting for a condition to be fulfilled). The transition from active to delayed is called the delaying of the process, the transition from delayed to active is called the re-activation of the process.

8.2 MUTUAL EXCLUSION AND REGIONS

8.2.1 GENERAL

Regions (see section 7.7) are a means of providing processes with mutually exclusive access to locations declared in them. Static context conditions (see section 8.2.2) are made such that accesses by a process (which is not the imaginary outermost process) to locations declared in a region can only be made by calling procedures which are defined inside the region and granted by the region.

A procedure name is said to denote a critical procedure (and it is a critical procedure name) if and only if it is defined inside a region and granted by the region, or if a procedure name with the same procedure definition (see section 7.4) is critical (the latter becomes relevant only when entry definitions are involved).

A region is said to be free if and only if control lies in none of its critical procedures nor in the region itself performing reach-bound initialisations.

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 will be released, becoming free again if:

- The region is left.
- The critical procedure returns.
- The critical procedure executes an action which causes the executing process to become delayed (see section 8.3). In the case of dynamically nested critical procedure calls, only the latest locked region will be released.

If, while the region is locked, a process attempts to call one of its critical procedures or attempts to enter the region, the attempting process is suspended until the region is released. (Note that the attempting process remains active in the CHILL sense).

When a region is released and more than one process has been suspended while attempting to enter the region or to call one of its critical procedures or to be re-activated in one of its critical procedures, only one process will be selected to enter the region according to an implementation defined scheduling algorithm.

8.2.2 *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 initialisations, the following static context conditions are enforced:

- the regionality requirements mentioned in the appropriate sections (assignment action, procedure call, send action, result action);
- regional procedures are not general (see section 7.4);
- critical procedures are neither general nor recursive (see section 7.3).

A *location*, *value* or *procedure name* can be regional. This property is defined as follows:

1. Location

   A *location* is regional if and only if any of the following conditions is fulfilled:

   - It is an *access name* that is either:
     - a *location name* declared textually inside a *region* and which is not defined in a *formal parameter* of a critical procedure,
     - a *loc-identity name*, where the *static mode location* in its declaration is regional or which is defined in a *formal parameter* of a regional procedure,
     - a *based name* where the *bound or free reference location name* in its declaration is regional,
     - a *location enumeration name*, where the *array location* in the associated *do action* is regional,
     - a *location do-with name*, where the *structure location* in the associated *do action* is regional.
   - It is a *dereferenced bound reference*, where the *bound reference expression* in it is regional.
   - It is a *dereferenced free reference*, where the *free reference expression* in it is regional.
   - It is a *dereferenced row*, where the *row expression* in it is regional.
   - It is an *array element*, *sub-array* or *array slice*, where the *array location* in it is regional.

- It is a *string element*, *substring* or *string slice*, where the *string location* in it is regional.

- It is a *structure field*, where the *structure location* in it is regional.

- It is a *location procedure call*, where in the *location procedure call* a *procedure name* is specified which is regional.

- It is a *location built-in routine call*, that the implementation specifies it is regional.

- It is a *location conversion*, where the *static mode location* in it is regional.

2. Value

A *value* or *expression* is regional if and only if it is either a *primitive value* which is regional or a *parenthesised expression* containing an *expression* which is regional.

A *primitive value* is regional if and only if any of the following conditions is fulfilled:

- It is a *location contents* which is regional and whose mode has the referencing property.

- It is a *value name* which is either a:

  - *synonym name*, where the *constant value* in its definition is regional,

  - *value do-with name*, where the *structure expression* in the associated *do action* is regional and whose mode has the referencing property.

- It is a *tuple* containing an *array tuple* or *structure tuple* in which at least one of the specified *value* occurrences is regional.

- It is a *value array element*, a *value sub-array*, or a *value array slice*, where the *array expression* in it is regional and the element mode of the mode of the *array expression* has the referencing property.

- It is a *value structure field*, where the *structure expression* in it is regional and the mode of the field has the referencing property.

- It is a *referenced location*, where the *location* in it is regional.

- It is an *expression conversion*, where the *expression* in it is regional.

- It is a *value procedure call*, where in the *value procedure call* a *procedure name* is specified which is regional and whose result mode has the referencing property.

- It is a *value built-in routine call*, which is either an *implementation value built-in routine call* which returns a value whose class is compatible with a mode which has the referencing property. and for which the implementation specifies that it is regional, or *ADDR(<location>)*, where *location* is regional.

3. Procedure name

   A *procedure name* is regional if and only if it is defined inside a *region* and it is not critical (i.e. not granted by the region).

8.3 DELAYING OF A PROCESS

When a process is active, it can become delayed by executing or evaluating one of the following actions or expressions:

Delay action (see section 6.16). When a process executes a delay action, it becomes delayed. It becomes a member with a priority of a set of delayed processes attached to the specified event location.

Delay case action (see section 6.17). When a process executes a delay case action, it becomes delayed. It becomes a member, with the specified priority, of each set of delayed processes that is attached to an event location specified in a delay alternative of the delay case action.

Receive expression (see section 5.2.18). When a process evaluates a receive expression, it becomes delayed if and only if there are no values in, nor sending processes delayed on the specified buffer location. It becomes a member of a set of delayed receiving processes attached to the specified (empty) buffer location.

Receive buffer case action (see section 6.19.3). When a process executes a receive buffer case action it becomes delayed if and only if in none of the specified buffer locations a value is present, no sending process is delayed on any of the specified buffer locations, and if *ELSE* is not specified. It becomes a member of each set of delayed receiving processes that is attached to a buffer location specified in a buffer-receive alternative of the receive buffer case action.

Receive signal case action (see section 6.19.2). When a process executes a receive signal case action, it becomes delayed if and only if no signal which may be received by the process executing the receive signal case action is pending and only if *ELSE* is not specified. The process becomes a member of each set of delayed processes attached to a signal name specified in the signal-receive alternative.

Send buffer action (see section 6.18.3). When a process executes a send buffer action, it becomes delayed if and only if the mode of the buffer location has a length attached and the number of values in the buffer is equal to the length just prior to the sending operation. The process becomes a member, with the specified priority, of the set of delayed sending processes attached to the buffer location.

When a process executes an action which causes it to become delayed while its control lies within a critical procedure, the associated region will be released. The dynamic context of the procedure will be retained until the process is re-activated where is was delayed in the region. The region will then be locked again.

8.4 RE-ACTIVATION OF A PROCESS

When a process is delayed, it can become re-activated if and only if another process executes one of the following actions:

Continue action (see section 6.15). When a process executes a continue action, it re-activates another process if and only if the set of delayed processes of the specified event location is not empty. A process of the highest priority is selected to become active according to an implementation defined scheduling algorithm. This re-activated process is thus removed from all sets of delayed processes.

Send buffer action (see section 6.18.3). If a process executes a send buffer action, it re-activates another process if and only if the set of delayed receiving processes of the specified buffer location is not empty. A process is selected to become active according to an implementation defined scheduling algorithm. This re-activated process is thus removed from all sets of delayed processes. If the set of delayed receiving processes of a specified buffer location is empty, the sent value will be stored into the buffer with its specified priority if the buffer capacity allows for it (see section 8.3).

Send signal action (see section 6.18.2). When a process executes a send signal action, it re-activates another process if and only if the set of delayed processes of the specified signal name contains a process that may receive the signal. A process is selected to become active according to an implementation defined scheduling algorithm. This re-activated process is thus removed from all sets of delayed processes. If no delayed process is present to receive the signal, the signal becomes pending, with its specified priority, possible list of values, process name and/or instance value.

Receive buffer case action (see section 6.19.3). When a process executes a receive buffer case action, it re-activates another process if and only if the set of delayed sending processes of any of the specified buffer locations is not empty. In that case it receives a value of the highest priority among the values in the buffer location or the delayed sending processes. Receiving a value from a buffer, the process removes the value

from the buffer and a delayed sending process with the value of the highest priority is selected to become active according to an implementation defined scheduling algorithm. This re-activated process is thus removed from all sets of delayed sending processes and its value is stored in the buffer, with the specified priority. Receiving a value directly from a delayed sending process, the delayed process carrying the value with the highest priority is selected to become active according to an implementation defined algorithm. This re-activated process is thus removed from all sets of delayed sending processes and its value is received.

When a process executes an action which causes another process to become active, while the re-activating process is active within a critical procedure, the re-activating process will remain active, i.e. it will not release the region at that point.

8.5 SIGNAL DEFINITION STATEMENTS

syntax:

<signal definition statement> ::= (1)
SIGNAL <signal definition> {,<signal definition>};* (1.1)

<signal definition> ::= (2)
<name> [= (<mode> {,<mode>})] [TO <process name>]* (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 name is a signal name if and only if it is defined in a *signal definition*. A signal name has the following properties:

- It has an optional list of modes attached, which are the modes mentioned in the *signal definition*.
- It has an optional process name attached which is the *process name* specified after *TO*.

static conditions: No *mode* in a *signal definition* may have the synchronisation property.

examples:

```
15.16   SIGNAL INITIATE = (INSTANCE),
                  TERMINATE;                    (1.1)
```

9.0 GENERAL SEMANTIC PROPERTIES

9.1 MODE CHECKING

9.1.1 PROPERTIES OF MODES AND CLASSES

9.1.1.1 Novelty

Informal

The novelty of a mode indicates whether or not it is defined via a newmode definition statement. The novelty of a mode is either nil, i.e. it is a (base) mode not defined via a newmode, or it is the newmode name via which it is defined.

Definition

The novelty of a mode is defined as follows:

- If the mode is denoted by a *newmode name*, its novelty is that newmode name,
- else if the mode is denoted by a *synmode name*, its novelty is the novelty of the *defining mode* in its definition,
- else if the mode is denoted by a *parameterised array mode*, *parameterised string mode*, or *parameterised structure mode*, its novelty is the novelty of the *origin array mode name*, *origin string mode name* or *origin variant structure mode name*, respectively, in it,
- else if the mode is denoted by a *range mode*, its novelty is the novelty of its parent mode,
- else if the mode is denoted by a virtually introduced parent mode, its novelty is the newmode name which caused its introduction (see section 3.2.3),
- else if the mode is denoted by *READ* ⟨*mode*⟩, its novelty is the novelty of the ⟨*mode*⟩,
- otherwise the novelty is nil.

9.1.1.2 Read-only modes

Informal

A mode is said to be read-only if a location of that mode, as a whole, is read-only, i.e. neither it nor any part of it may be overwritten.

Definition

A mode has the following hereditary property: it is a read-only mode if and only if any of the following conditions is fulfilled:

- It is denoted by a *mode* which is of the form *READ <mode>*.
- It is denoted by a *parameterised array mode*, a *parameterised string mode* or a *parameterised structure mode*, where the *origin array mode name*, *origin string mode name* or *origin variant structure mode name*, respectively, in it denotes a read-only mode.

9.1.1.3 Read-only property

Informal

A mode has the read-only property if a location of that mode is read-only or contains a component or a sub-component etc. which is read-only.

Definition

A mode has the read-only property if and only if one of the following conditions is fulfilled:

- The mode is a mode name, defined by a mode which has the read-only property.
- The mode is an array mode with an element mode which has the read-only property or a structure mode where at least one of its field modes has the read-only property.
- The mode is a read-only mode.

9.1.1.4 Referencing property

Informal

A mode has the referencing property if a location of that mode has a reference mode or contains a component or a sub-component etc. which has a reference mode.

Definition

A mode has the referencing property if and only if one of the following conditions is fulfilled:

- The mode is a mode name defined by a mode which has the referencing property.
- The mode is an array mode with an element mode which has the referencing property or a structure mode where at least one of its field modes has the referencing property.
- The mode is a reference mode.

9.1.1.5 Tagged parameterised property

Informal

A mode has the tagged parameterised property if a location of that mode has a tagged parameterised structure mode or contains a component or a sub-component etc. which has a tagged parameterised structure mode.

Definition

A mode has the tagged parameterised property if and only if one of the following conditions is fulfilled:

- The mode is a mode name defined by a mode which has the tagged parameterised property.
- The mode is an array mode with an element mode which has the tagged parameterised property or a structure mode where at least one of its field modes has the tagged parameterised property.
- The mode is a tagged parameterised structure mode.

9.1.1.6 Synchronisation property

Informal

A mode has the synchronisation property if a location of that mode has a synchronisation mode or contains a component or a sub-component etc. which has a synchronisation mode.

Definition

A mode has the synchronisation property if and only if one of the following conditions is fulfilled:

- The mode is a mode name defined by a mode which has the synchronisation property.
- The mode is an array mode with an element mode which has the synchronisation property or a structure mode where at least one of its field modes has the synchronisation property.
- The mode is an event mode or a buffer mode.

9.1.1.7 Root mode

Any M-value class or M-derived class, where M is not a composite mode, has a root mode defined as:

- if M is not a range mode then the root mode is M,
- if M is a range mode then the root mode is the parent mode of M.

9.1.1.8 Resulting class

Given two compatible classes (see section 9.1.2.6), which are either the all class, an M-value class or an M-derived class, where M is either a discrete mode, a powerset mode or a string mode, the resulting class is defined as:

- the resulting class of the M-derived class and the N-derived class is the M-derived class;
- the resulting class of the M-value class and the N-derived class, is if M is not a range mode then the M-value class, otherwise the P-value class, where P is the parent mode of M;
- the resulting class of the M-value class and the N-value class is , if M is not a range mode then the M-value class, otherwise the P-value class, where P is the parent mode of M;
- the resulting class of the all class and any other class is the latter class.

Given a list C_i of pairwise compatible classes (i=1,...,n), the resulting class of the list of classes is recursively defined as, if n>1 then as the resulting class of the resulting class of the list C_i (i=1,...,n-1) and the class C_n, otherwise as the resulting class of C_1 and C_1.

(Note that CHILL is defined in such a way that the order of taking the classes C_i is irrelevant, i.e. all such resulting classes are compatible.)

9.1.2 *RELATIONS ON MODES AND CLASSES*

In the following sections, the compatibility relations are defined between modes, between classes, and between modes and classes. These relations are used throughout the document to define static conditions.

The compatibility relations themselves are defined in terms of some other relations which are mainly used in chapter 9 for the above mentioned purpose.

9.1.2.1 The relation "defined by"

Informal

A mode name is said to be defined by its defining mode and, transitively, if the latter is also a mode name, the former is also defined by the defining mode of its defining mode etc.

Definition

A mode name N is said to be defined by a mode M if and only if:

- M is the defining mode of N
- the defining mode of N is a mode name defined by M.

9.1.2.2 Equivalence relations on modes

GENERAL

Informal

The following equivalence relations play a role in the formulation of the compatibility relations:

- Two modes are said to be similar if they are of the same kind, i.e. they have the same hereditary properties.
- Two modes are said to be v-equivalent (value-equivalent) if they are similar and also have the same novelty.

- Two modes are said to be 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 said to be l-equivalent (location-equivalent) if they are equivalent and also have the same read-only specification.

Definition

In the following sections, 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 sections).

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. The modes are equivalent 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).

The relation "similar"

Two modes are similar if and only if one of the following conditions is fulfilled:

- they are integer modes;

- they are boolean modes;

- they are character modes;

- they are set modes such that they define the same number of values, the same set element names and for the same names, the *NUM* built-in routine call delivers the same value;

- they are range modes with similar parent modes;

- the one is a range mode whose parent mode is similar to the other mode;

- the one is a boolean mode and the other a bit string mode of length 1;

- the one is a character mode and the other a character string mode of length 1;

- 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, same parameter attributes and, if present, the same register specification;

  2. they both have or both do not have a result spec. If present, both result specs must have l-equivalent modes, the same attributes and the same register specification, if present;

  3. they have the same set of exception names;

  4. they have the same recursivity;

- they are instance modes;

- they are event modes such that they both have no length or the same length;

- they are buffer modes such that:

  1. they both have no length or the same length;

  2. they have l-equivalent buffer element modes;

- they are string modes such that:

  1. they both are bit string modes or character string modes;

  2. they have the same length. This check is dynamic in the case that one or both modes is (are) dynamic. Check failure will result in the *RANGEFAIL* exception;

- 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 (see section 9.1.2.2);

  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 parameterised structure modes such that:

  1. they have the same number of fields and corresponding (by position) fields are equivalent (see section 9.1.2.2);

  2. if they are both parameterisable variant structure modes, their lists of classes must be compatible;

- they are parameterised structure modes such that:

  1. their origin variant structure modes are similar;

  2. their corresponding (by position) values must be the same. This check is dynamic if one or both modes is (are) dynamic. Check failure will result in the *TAGFAIL* exception.

The relation "v-equivalent"

Two modes are v-equivalent if and only if they are similar and have the same novelty.

The relation "equivalent"

Two modes are equivalent if and only if they are v-equivalent and:

- if the one mode is a boolean mode, the other mode must also be a boolean mode;

- if the one mode is a character mode, the other mode must also be a character mode;

- if the one mode is a range mode, the other mode must also be a range mode and both upper bounds must be equal and both lower bounds must be equal.

The relation "l-equivalent"

Two modes are l-equivalent if and only if they are equivalent and if the one mode has the read-only property, the other mode must also have the read-only property, and:

- if both are bound reference modes, their referenced modes must be l-equivalent;

- if both are row modes, their referenced origin modes must be l-equivalent;

- if both are array modes, their element modes must be l-equivalent;

- if both are structure modes corresponding (by position) fields must be l-equivalent.

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:

1. Fixed fields and variant fields

   a. Both fields must have equivalent layout.

   b. Both field modes must be 1. equivalent, 2. l-equivalent.

2. Alternative fields

   a. Both alternative fields have tags or both have no tags. In the former case, the tags must have the same number of tag field names and corresponding (by position) tag field names must denote corresponding fixed fields.

   b. Both must have the same number of variant alternatives and corresponding (by position) variant alternatives must be 1. equivalent, 2. l-equivalent.

   c. 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.

3. Variant alternatives

   a. 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 *(ELSE)*, or both define the same set of values.

   b. Both variant alternatives must have the same number of variant fields and corresponding (by position) variant fields must be 1. equivalent, 2. l-equivalent.

The relation "equivalent" for layout

In the sequel, it will be assumed that each *pos* is of the form:
POS(<word number>,<start bit>,<length>)

and that each *step* is of the form:
STEP(<pos>,<step size>,<pattern size>)

Section 3.10.6 gives the appropriate rules to bring *pos* or *step* in the required form.

1. 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).

2. 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 *NUM(step size)* must deliver the same values for the two element layouts and *NUM(pattern size)* must deliver the same values for the two element layouts.

3. Pos

   A *pos* is equivalent to another *pos* if and only if both *NUM(word number)* occurrences deliver the same value, both *NUM(start bit)* occurrences deliver the same value and both *NUM(length)* occurrences deliver the same value.

9.1.2.3 The relation "read-compatible"

Informal

A mode M is said to be read-compatible with a mode N if and only if M and N are equivalent and M and its possible (sub-)components have more restrictive read-only specifications. This relation is therefore non-symmetric.

Example:
READ REF READ CHAR is read-compatible with *REF 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 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 read-compatible with the element mode of N;

- if M and N are structure modes, any field mode of M must be read-compatible with the corresponding field mode of N.

9.1.2.4 The relation "restrictable to"

Informal

The relation "restrictable to" 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 equally or less restrictive read-only specification than those referenced by N. This relation is therefore non-symmetric. The relation is used in assignments (see section 9.1.2.5).

Example:
REF INT is restrictable to *REF READ INT*
STRUCT(P REF BOOL) is restrictable to *STRUCT(Q REF READ BOOL)*

Definition

A mode M is restrictable to a mode N (a non-symmetric relation) if and only if one of the following holds:

- M does not have the referencing property and M is equivalent to N.

- M and N are bound reference modes and the referenced mode of N is read-compatible with the referenced mode of M.

- M and N are free reference modes and M and N are equivalent.

- M and N are row modes and the referenced origin mode of N is read-compatible with the referenced origin mode of M.

- M and N are array modes and the element mode of M is restrictable to the element mode of N.

- M and N are structure modes and each field mode of M is restrictable to the corresponding field mode of N.

9.1.2.5 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 it is a reference mode and one of the following conditions is fulfilled:
  1. N is a static mode and M is a bound reference mode whose referenced mode is read-compatible with N;
  2. N is a static mode and M is a free reference mode;
  3. M is a row mode with referenced origin mode V and:
     - if V is a string mode, N must be a string mode such that V(p) is read-compatible with N, where p is the (possibly dynamic) length of N;
     - if V is an array mode, N must be an array mode such that V(p) is read-compatible with N, where p is the (possibly dynamic) upper bound of N;
     - if V is a variant structure mode, N must be a parameterised structure mode such that $V(p_1,...p_n)$ is read-compatible with N, where $p_1,...p_n$ denote the list of values of 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, N must be restrictable to M.

9.1.2.6 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-derived class or 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 location and N is a free reference mode.

  3. N is a row mode with referenced origin mode V and:

     - if V is a string mode, M must be a string mode such that V(p) is equivalent to M, where p is the (possibly dynamic) length of M;

     - if V is an array mode, M must be an array mode such that V(p) is equivalent to M, where p is the (possibly dynamic) upperbound of M;

     - if V is a variant structure mode, M must be a parameterised structure mode such that $V(p_1,...p_n)$ is equivalent to M, where $p_1,...p_n$ denote the list of values of N.

- 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.

9.1.3 *CASE SELECTION*

syntax:

```
<case label specification> ::=                          (1)
    <case label list> {,<case label list>}*             (1.1)

<case label list> ::=                                   (2)
    (<case label> {,<case label>}*)                     (2.1)
  | (ELSE) | <irrelevant>                               (2.2)
```

```
<case label> ::=                                        (3)
     <discrete literal expression>                      (3.1)
   | <literal range>                                    (3.2)
   | <discrete mode name>                               (3.3)

<irrelevant> ::=                                        (4)
     (*)                                                (4.1)
```

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 section 3.10.4), in which case a list of variant fields is selected,
- labelled array tuples (see section 5.2.5), in which case an array element value is selected,
- case action (see section 6.4), in which case an action statement list is selected.

In the first and last situation, 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,
- the case label list consists of *(ELSE)* and the value is one of the values implicitly indicated by *(ELSE)*
- the case label list consists of *irrelevant*.

The values explicitly indicated by a case label are the values delivered by any *discrete 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*, 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*.

- A *case label list* has a class attached, which is, if it is *(ELSE)* or *<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 (provided that the case label specifications have the same number of classes; this will always be the case). This resulting list of classes is formed by forming, 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 section 3.10.4).

- 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, 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.

examples:

```
11.7     (occupied)                    (3.1)
11.62    (rook),(*)                    (1.1)
8.24     (ELSE)                        (2.2)
```

9.1.4 *DEFINITION AND SUMMARY OF SEMANTIC CATEGORIES*

This section 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 sections, the definition is given here, otherwise the appropriate section will be referenced.

9.1.4.1 Names

Mode names

array mode name:	a *name* defined by an array mode.
boolean mode name:	a *name* defined by a boolean mode.
bound reference mode name:	a *name* defined by a bound reference mode.
buffer mode name:	a *name* defined by a buffer mode
character mode name:	a *name* defined by a character mode.
discrete mode name:	a *name* defined by a discrete mode.
event mode name:	a *name* defined by an event mode.
free reference mode name:	a *name* defined by a free reference mode.
instance mode name:	a *name* defined by an instance mode.

integer mode name:	a *name* defined by an integer mode.
mode name:	see section 3.2.1
newmode name:	see section 3.2.3
parameterised array mode name:	a *name* defined by a parameterised array mode.
parameterised string mode name:	a *name* defined by a parameterised string mode.
parameterised structure mode name:	a *name* defined by a parameterised structure mode.
powerset mode name:	a *name* defined by a powerset mode.
procedure mode name:	a *name* defined by a procedure mode.
range mode name:	a *name* defined by a range mode.
row mode name:	a *name* defined by a row mode.
set mode name:	a *name* defined by a set mode.
string mode name:	a *name* defined by a string mode.
structure mode name:	a *name* defined by a structure mode.
synmode name:	see section 3.2.2
variant structure mode name:	a *name* defined by a variant structure mode.

Access names

based name:	see section 4.1.4
location name:	see sections 4.1.2, 7.4
location do-with name:	see section 6.5.4
location enumeration name:	see section 6.5.2
loc-identity name:	see sections 4.1.3, 7.4

Value names

synonym name:	see section 5.1
value do-with name:	see section 6.5.2
value enumeration name:	see section 6.5.4
value receive name:	see sections 6.19.2, 6.19.3

Miscellaneous names

bound or free reference location name:	a *location name* with a bound reference mode or a free reference mode.
built-in routine name:	any implementation defined name denoting an implementation defined built-in routine.
field name:	see section 3.10.4
general procedure name:	a *procedure name* whose generality is general.
label name:	see section 6.1
module name:	see section 7.6

non-reserved name:	a *name* which is none of the reserved names mentioned in Appendix C1.
procedure name:	see section 7.4
process name:	see section 7.5
region name:	see section 7.7
register name:	an implementation defined name denoting a machine register.
reserved name list:	a *name list* consisting solely of reserved names. (see Appendix C1)
set element name:	see section 3.4.5
signal name:	see section 8.5.2
tag field name:	see section 3.10.4
undefined synonym name:	see section 5.1

9.1.4.2 Locations

array location:	a *location* with an array mode.
buffer location:	a *location* with a buffer mode.
event location:	a *location* with an event mode.
instance location:	a *location* with an instance mode.
string location:	a *location* with a string mode.
structure location:	a *location* with a structure mode.

9.1.4.3 Expressions

array expression:	an *expression* whose class is compatible with an array mode.
boolean expression:	an *expression* whose class is compatible with a boolean mode.
bound reference expression:	an *expression* whose class is compatible with a bound reference mode.
discrete expression:	an *expression* whose class is compatible with a discrete mode.
discrete literal expression:	a *discrete expression* which is literal.
free reference expression:	an *expression* whose class is compatible with a free reference mode.
instance expression:	an *expression* 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.

powerset expression: an *expression* whose class is compatible with a powerset mode.

procedure expression: an *expression* whose class is compatible with a procedure mode.

row expression: an *expression* whose class is compatible with a row mode.

string expression: an *expression* whose class is compatible with a string mode.

structure expression: an *expression* whose class is compatible with a structure mode.

static conditions: Neither a *boolean expression* nor a *discrete expression* (when indicated in the syntax) may have a dynamic class. I.e. the check whether the *expression* is compatible with a boolean mode or a discrete mode, can be made statically.

9.1.4.4 Miscellaneous semantic categories

implementation value built-in routine call: see section 11.1.3

location procedure call: see section 6.7

module action statement: an *action statement* in which the directly contained *action* is a *module*.

non-apostrophe character: a *character* which is not an apostrophe.

value procedure call: see section 6.7.

9.2 VISIBILITY AND NAME BINDING

9.2.1 GENERAL

The specific CHILL constructs mentioned in section 7.1 create new names within a program. The program structuring statements and visibility statements determine the visibility of names throughout the program. This section deals with the visibility of names with the exclusion of exception names, i.e. each name is considered not to be in the context of an *exception name* . See chapter 10 for exception names.

To enable a precise description of the visibility structure of a program, the following refinements of terminology are introduced just for this section 9.2:

- A name string (of a name) is a string of characters (used as denotation for the name) seen as a lexical element isolated from any context. A name is a name string associated with a definition (defining occurrence, see section 9.2.2) of that name string.

Example:

```
B: BEGIN
   MODULE DCL I INT; END;
   MODULE DCL I PTR; END;
   END B;
```

In the begin-end body of the block labelled *B* two names are introduced, both with the name string *I*.

Within a reach, each name has one of the following four degrees of visibility:

Table 1.Degrees of visibility

Visibility	Properties (informal)
directly strongly visible	Name is visible by creation, granting or seizing
indirectly strongly visible	Name is inherited via block nesting or by its pervasive attribute
weakly visible	Name is implied by a strongly visible name
Invisible	Name may not be applied

A name is said to be strongly visible if it is either directly strongly visible or indirectly strongly visible. A name is said to be visible if it is either weakly or strongly visible,otherwise the name is said to be invisible. The program structuring statements and visibility statements determine uniquely to which visibility class each name belongs. The precise properties of the visibility classes are explained in the following sections.

Name binding is the mechanism of associating a unique name to any name string, i.e. associating a unique meaning to the name string.

9.2.2 *VISIBILITY AND NAME CREATION*

Names are created by the constructs mentioned in section 7.1. Except for field names and set element names, the names have a unique defining occurrence, which is the construct that introduces the name. In order to have a uniform treatment for any name for establishing the visibility and name binding, the following mechanism for giving a unique defining occurrence to any created name is considered to be applied:

- Within the reach of a group, each *mode* occurrence is considered to be an applied occurrence of a virtual synmode name defined within that reach. For procedure definitions, the virtual synmode definition of the result mode is placed in the reach of the group surrounding the procedure. The virtual synmode definitions of the formal parameter modes are placed in the reach of the procedure.

Visibility and name binding rules are applied taking these virtual definitions into account.

Example:

```
DCL I SET(A,B),
    K INT,
    J ARRAY (SET(A,B)) INT;
```

is considered to be replaced by:

```
SYNMODE &1 = SET(A,B), &2 = INT;
        &3 = ARRAY (&1)&2;
DCL I &1, K &2, J &3;
```

&1, *&2* and *&3* are virtual synmode names. The visibility rules are applied to these virtual replacements. The virtual replacements have the consequence that name creating modes (*SET*, *STRUCT*) appear only once in a reach, at the right-hand side of a virtual synmode definitions. This synmode definition is considered to be an unique defining occurrence of the set element names or field names.

The visibility and name binding properties of field names are different (simpler) than those of other names. Therefore, in the remainder of section 9.2, the word "name" does not include field names, unless otherwise stated.

9.2.3 *IMPLIED NAMES*

Each strongly visible name in a reach has a (possibly empty) set of implied names defined as follows;

- Each mode has a (possibly empty) set of implied names, listed in Table 2.

The implied names of a (strongly visible) name are:

- If the name is an access name, the implied names are the names implied by the mode of the access name.
- If the name is a mode name, the implied names are the names implied by the defining mode.
- If the name is a procedure name, the implied names are the names implied by the mode of the result spec.
- If the name is a signal name, the implied names are all names implied by its attached modes.
- Otherwise there are no implied names.

Table 2. Implied names of modes

Modes	Set of implied names
INT, BIN, CHAR *INSTANCE, PTR* *BOOL, EVENT* *CHAR(n), BIN(n)* *BIT(n), RANGE(....)*	Empty
mode name	The set of implied names of its defining mode
M(m:n)	The set of implied names of *M*
REF M, ROW M *READ M, POWERSET M* *PROC(---) (M)* *BUFFER M*	The set of implied names of *M*
ARRAY (M) N	The union of the sets of the names implied by *M* and *N*
STRUCT(N_1 M_1,---,N_n M_n)	The union of the sets of names implied by M_1 through M_n. For variant structures it is the union of the implied names of all the fields of the variant structure
Parameterised *VM(---)*	The set of implied names of *VM*
SET(.....)	The set of set element names

(note that implied names, always being set element names, never have implied names themselves.)

9.2.4 *VISIBILITY IN REACHES*

A name, which is strongly visible in a reach, is either directly strongly visible or indirectly strongly visible in it.

A name is directly strongly visible in a reach only in the following cases:

- The name has its defining occurrence in the reach.
- The name is seized into the reach (see section 9.2.6.3).

- The name is granted into the reach (see section 9.2.6.2).

A name is indirectly strongly visible in a reach only in the following cases:

- The reach is a block reach and the name is inherited (see section 9.2.5).

- The name is strongly visible in the surrounding reach and has the pervasive property in that surrounding reach (see section 9.2.6.2) and the reach has no directly strongly visible name in it with the same name string. In this case, the name has also the pervasive property in the reach.

A name is weakly visible in a reach only in the following case:

- The name is implied by a strongly visible name in that reach.

The visibility rules are defined such that in any reach, all the strongly visible names have different name strings. However, two or more weakly visible names may have the same name string. Such a name may then not be applied in some cases (see section 9.2.8).

9.2.5 *VISIBILITY AND BLOCKS*

The following visibility rule applies to blocks:

- A name, strongly visible in the reach of a group, is indirectly strongly visible in the reach of each directly enclosed block which has no directly strongly visible name with the same name string (name inheritance by blocks).

9.2.6 *VISIBILITY AND MODULIONS*

9.2.6.1 General

syntax:

```
<visibility statement> ::=                    (1)
    <grant statement>                         (1.1)
  | <seize statement>                         (1.2)
```

semantics: Visibility statements, which are only allowed in modulion reaches, control the visibility of the names explicitly mentioned in them (and implicitly their implied names).

9.2.6.2 Grant statements

syntax:

```
<grant statement> ::=                                          (1)
     GRANT <grant window> [PERVASIVE];                         (1.1)

<grant window> ::=                                             (2)
     <granted element> {,<granted element>}*                   (2.1)
   | ALL                                                       (2.2)

<granted element> ::=                                          (3)
     <non reserved name>                                       (3.1)
   | <newmode name> <forbid clause>                            (3.2)

<forbid clause> ::=                                            (4)
     FORBID {<forbid name list> | ALL}                         (4.1)

<forbid name list> ::=                                         (5)
     (<field name> {,<field name>}*)                           (5.1)
```

semantics: Grant statements are means of extending the visibility of names in a modulion reach into the directly surrounding reach. *FORBID* can only be specified 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 module, not outside.

The following visibility rules apply:

- A name, visible in the reach of a modulion, is directly strongly visible in the reach of the directly surrounding group if it is mentioned in a grant statement in the modulion reach. The name is said to be granted into the surrounding reach.

- The notation *FORBID ALL* is a syntactic shorthand forbidding all the field names of the *newmode name* (see section 9.2.7).

- The notation *GRANT ALL [PERVASIVE]* is a syntactic shorthand for granting all the names (with the pervasive property, if specified), which are strongly visible in the reach of the granting modulion and whose defining occurrence lies inside the granting modulion.

static properties: A name granted with the attribute *PERVASIVE*, has the pervasive property in the surrounding reach.

static conditions: The defining occurrence of any *non-reserved name* must lie inside the granting modulion.

The *newmode name* with *FORBID* specification must have its defining occurrence in the reach of the granting modulion and must be a structure mode and each *field name* in the *forbid name list* must be a field name of the *newmode name*.

If a grant statement is placed in the reach of a region it must not grant a name which is a regional value name or a regional access name.

examples:

```
1.11    GRANT add,mult;                                    (1.1)
```

9.2.6.3 Seize statements

syntax:

```
<seize statement> ::=                                      (1)
     SEIZE <seize window>;                                 (1.1)

<seize window> ::=                                         (2)
     <seized element> {,<seized element>}*                 (2.1)
   | ALL                                                   (2.2)

<seized element> ::=                                       (3)
     <modulion name> ALL                                   (3.1)
   | <non-reserved name>                                   (3.2)

<modulion name> ::=                                        (4)
     <module name>                                         (4.1)
   | <region name>                                         (4.2)
```

semantics: Seize statements are a means of extending the visibility of names in group reaches into the reaches of directly enclosed modulions.

The following visibility rules apply:

- A name, visible in the reach of a group, is directly strongly visible in the reach of a directly enclosed modulion if it is mentioned in a seize statement in the modulion reach. The name is said to be seized in the modulion reach.

- If a name which has the pervasive property in the surrounding reach is seized, it will be directly strongly visible in the reach of the seizing module and it keeps the pervasive property.

- The notation *SEIZE ALL* is a syntactic shorthand for seizing all the names which are strongly visible in the reach of the surrounding group and whose defining occurrence lies outside the seizing modulion.

- The notation *SEIZE <modulion name> ALL* is a syntactic shorthand for seizing all the names which are strongly visible in the reach of the surrounding group and are granted by the module or region denoted by the *modulion name*.

static conditions: The defining occurrence of any *non-reserved name* or *modulion name* must lie outside the seizing modulion.

A name mentioned in a *seized element* must not be a *value do-with name* nor a *location do-with name*.

examples:

```
15.14   SEIZE /*external signals */
        ACQUIRE, RELEASE, CONGESTED, STEP, READOUT;          (1.1)
```

9.2.7 *VISIBILITY OF FIELD NAMES*

Field names may occur outside their defining occurrence only in the following context:

- Field selection of a structure location or a structure value.
- Labelled structure tuples.
- Forbid clauses in the grant statement.

In the first two contexts those field names which are attached to the mode of the structure location, (strong) structure value or tuple are visible, except if the novelty of this mode is a newmode name which has been granted by a modulion with a forbid clause. In the latter case, outside the granting modulion, only those field names which are not mentioned in the *forbid clause* are visible.

In the last context all and only the field names of the granted newmode name are visible.

9.2.8 *NAME BINDING*

Name binding is the mechanism of associating a unique name with any occurrence of a name string.

The binding rules depend on whether the name string occurs in the context of:

1. a *directive name*,

2. an *exception name*,

3. a *reserved name*,

4. a *field name*,

5. a *non-reserved name*, a *module name*, or *region name* in a *seized element*,

6. any other *name*.

Binding rules

1. A directive name string follows an implementation defined binding scheme which must not influence the CHILL binding rules (see section 2.6).

2. An exception name string is treated according to the handler identification rules given in section 10.3.

3. A reserved name string which is not freed by a free directive in a compilation unit in which it occurs, has its reserved meaning. If it is freed, it follows the rules under 6. Even if freed in a compilation unit, it may not be granted outside that unit.

4. A field name string is bound as follows, depending upon the contexts mentioned in section 9.2.7:

   - to the visible field name of the structure mode of the structure location or (strong) structure expression;

   - to the visible field name of the structure mode of the (strong) tuple;

   - to the visible field name of the newmode name.

   If the name string cannot be bound to such a field name, the program is in error.

5. A name string occurring in the context of a *seized element* is bound according to the rules mentioned under 6., but in the reach directly surrounding the reach in which the seize statement is placed.

6. For any other occurrence of a name string in the reach of a group:

   a. if there is more than one strongly visible name in the reach with that name string, then the program is in error;

   b. else if there is one strongly visible name in the reach with that name string, then the name string is bound to that name;

   c. else if there is exactly one weakly visible name with that name string in the reach, the name string is bound to that name;

d. else if there is more than one weakly visible name in the reach with that name string and if all those (set element) names have compatible classes, then the name string is bound to (an arbitrary) one of those names;

e. otherwise the program is in error.

In addition to the rules mentioned above, a name string appearing in a grant statement or a seize statement must be bound to a name whose defining occurrence lies inside or outside, respectively, the granting or seizing modulion (if there is a choice according to rule d.).

10.0 EXCEPTION HANDLING

10.1 GENERAL

An exception is either a language defined exception, in which case it may have a language defined 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 named 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.

10.2 HANDLERS

syntax:

<handler> ::= (1)
*ON {<on-alternative>}**
[ELSE <action statement list>] END (1.1)

<on-alternative> ::= (2)
(<exception list>) : <action statement list> (2.1)

semantics: An action statement list in an on-alternative is entered if an exception occurs in the statement to which the handler is appended and whose name is mentioned in the *exception list* in the on-alternative. If *ELSE* is specified, the action statement list following it will be entered if an exception occurs in the statement to which the handler is appended and whose name is not specified in any *exception list* directly contained in the handler.

If the handler is appended to an action, when the end of an action statement list in an on-alternative is reached control will be given to the action statement following the action statement in which the handler is placed.

If the handler is appended to a procedure definition, control will be returned to the calling point when the end of an action statement list is reached. If the handler is appended to a process definition, the executing process will terminate

When the end of an action statement list in the on-alternative is reached.

static conditions: All the 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:

```
10.43   ON
          (no_space): CAUSE overflow;
        END                                              (1.1)
```

10.3 HANDLER IDENTIFICATION

When an exception E occurs at an action 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 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 E is determined as follows:

1. if a handler is appended to A or D which mentions E in an exception list or which specifies *ELSE*, then that handler is the appropriate one with respect to E;

2. otherwise, if A or D is directly enclosed by a bracketed action, the appropriate handler (if present) is the appropriate handler for the bracketed action with respect to E;

3. otherwise if A or D is placed in the reach of a procedure definition then:

   - if a handler is specified after the procedure definition which handler specifies E in an exception list or specifies *ELSE*, then that handler is the appropriate handler,

   - if E is mentioned in the exception list of the procedure definition then E is caused at the calling point,

   - otherwise there is no handler;

4. otherwise if A or D is placed in the reach of a process definition (possibly the imaginary one) then:

   - if a handler is specified after the process definition which handler specifies E in an exception list or specifies *ELSE* then that handler is the appropriate handler,

   - otherwise there is no handler;

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 definition D' with respect to E to which the handler is appended 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.

11.0 IMPLEMENTATION OPTIONS

11.1 IMPLEMENTATION DEFINED BUILT-IN ROUTINES

syntax:

```
<built-in routine call> ::=                                    (1)
     <built-in routine name>
     ([<built-in routine parameter list>])                     (1.1)

<built-in routine parameter list> ::=                          (2)
     <built-in routine parameter>
     {, <built-in routine parameter>}*                         (2.1)

<built-in routine parameter> ::=                               (3)
     <value>                                                   (3.1)
   | <location>                                                (3.2)
   | <non-reserved name>                                       (3.3)
```

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.

A value, a location or any program defined name which is not a reserved name may be passed as parameter. The built-in routine call may return a value or a location. The parameter passing mechanism is implementation defined.

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.

static properties: A *built-in routine name* is an implementation defined name which is considered to be defined in the standard prelude module (see section 7.8). It 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).

11.2 IMPLEMENTATION DEFINED INTEGER MODES

An implementation may define other integer modes 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 as newmode names, similar to *INT*. Their value ranges are implementation defined. These integer-modes may be defined as root modes of appropriate classes.

11.3 IMPLEMENTATION DEFINED REGISTER NAMES

An implementation may define a set of pre-defined register names (see sections 3.7 and 7.8).

11.4 IMPLEMENTATION DEFINED PROCESS NAMES AND EXCEPTION 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 standard prelude module. Processes of this name may be started and instance values denoting such processes may be manipulated.

An implementation may define a set of exception names for any process name or a group of process names. These exceptions may be caused when starting the process (see section 6.14).

11.5 IMPLEMENTATION DEFINED HANDLERS

An implementation may specify that an implementation defined handler is appended to the imaginary outermost process definition (see section 7.8). The exception names and actions in the implementation defined handler may specify any legal CHILL exception name or action. Note that an on-alternative in such handler can be entered only by an exception caused by the outermost process and not by any inner process.

11.6 SYNTAX OPTIONS

At some places, CHILL allows for more than one syntatic description for the same semantics. The choice for one of the following options should be fixed within the whole program.

Assignment symbol

The assignment symbol is either := or =

ARRAY

The reserved name *ARRAY* should be either mandatory or not allowed.

RETURNS

In procedure definitions with a result spec, the reserved name *RETURNS*, should be either mandatory or not allowed.

Structure modes

Structure modes must be either in the nested structure notation or in the level numbered notation.

Literal and tuple brackets

In the case that square brackets are available in the representation alphabet, the brackets *[* and *]* may be used instead of *(:* and *:)* respectively.

A.1 CCITT ALPHABET NO. 5 INTERNATIONAL REFERENCE VERSION

Recommendation V3 (The internal representation is the binary number formed by bits b7 to b1, where b1 is the least significant bit).

b_4	b_3	b_2	b_1		b_7 0	0	0	0	1	1	1	1
					b_6 0	0	1	1	0	0	1	1
					b_5 0	1	0	1	0	1	0	1
					0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	TC_7 (DLE)	SP	0	@	P		p
0	0	0	1	1	TC_1 (SOH)	DC_1	!	1	A	Q	a	q
0	0	1	0	2	TC_2 (STX)	DC_2	"	2	B	R	b	r
0	0	1	1	3	TC_3 (ETX)	DC_3	#	3	C	S	c	s
0	1	0	0	4	TC_4 (EOT)	DC_4	¤	4	D	T	d	t
0	1	0	1	5	TC_5 (ENQ)	TC_8 (NAK)	%	5	E	U	e	u
0	1	1	0	6	TC_6 (ACK)	TC_9 (SYN)	&	6	F	V	f	v
0	1	1	1	7	BEL	TC_{10} (ETB)	'	7	G	W	g	w
1	0	0	0	8	FE_0 (BS)	CAN	(	8	H	X	h	x
1	0	0	1	9	FE_1 (HT)	EM	)	9	I	Y	i	y
1	0	1	0	10	FE_2 (LF)	SUB	*	:	J	Z	j	z
1	0	1	1	11	FE_3 (VT)	ESC	+	;	K	[	k	{
1	1	0	0	12	FE_4 (FF)	IS_4 (FS)	,	<	L	\	l	|
1	1	0	1	13	FE_5 (CR)	IS_3 (GS)	-	=	M	]	m	}
1	1	1	0	14	SO	IS_2 (RS)	.	>	N	^	n	~
1	1	1	1	15	SI	IS_1 (US)	/	?	O	_	o	DEL

A.2 MINIMAL CHARACTER SET FOR REPRESENTING CHILL PROGRAMS

The following subset of the CCITT alphabet no. 5 Basic code is used in this document to represent CHILL programs.

b_4 b_3 b_2 b_1		b_7 0	0	0	0	1	1	1	1
		b_6 0	0	1	1	0	0	1	1
		b_5 0	1	0	1	0	1	0	1
			0	1	2	3	4	5	6	7
0 0 0 0	0			SP	0		P		
0 0 0 1	1				1	A	Q		
0 0 1 0	2				2	B	R		
0 0 1 1	3				3	C	S		
0 1 0 0	4				4	D	T		
0 1 0 1	5				5	E	U		
0 1 1 0	6				6	F	V		
0 1 1 1	7			'	7	G	W		
1 0 0 0	8			(	8	H	X		
1 0 0 1	9			)	9	I	Y		
1 0 1 0	10			*	:	J	Z		
1 0 1 1	11			+	;	K			
1 1 0 0	12			,	<	L			
1 1 0 1	13			-	=	M			
1 1 1 0	14			.	>	N			
1 1 1 1	15			/	?	O	_		

	Name	Use
;	semicolon	terminator for statements etc.
,	comma	separator in various constructs
(	left parenthesis	opening parenthesis of various constructs
)	right parenthesis	closing parenthesis of various constructs
[	left square bracket	opening bracket of a tuple
]	right square bracket	closing bracket of a tuple
(:	left tuple bracket	opening bracket of a tuple
:)	right tuple bracket	closing bracket of a tuple
:	colon	label indicator, range indicator
.	dot	field selection symbol
:=	assignment symbol	assignment, initialisation
<	less than	relational operator
<=	less than or equal	relational operator
=	equal	relational operator, assignment, initialisation
/=	not equal	relational operator
>=	greater than or equal	relational operator
>	greater than	relational operator
+	plus	addition operator
-	minus	subtraction operator
*	asterisk	multiplication operator, undefined value, unnamed value, irrelevant symbol
/	solidus	division operator
//	double solidus	concatenation operator
->	arrow	referencing and dereferencing
<>	diamond	start or end of a directive clause
/*	comment opening bracket	start of a comment
*/	comment closing bracket	end of a comment
'	apostrophe	start or end symbol in various literals
''	double apostrophe	apostrophe within character or character string literals
_	underline	spacer in names and literals

APPENDIX C: CHILL SPECIAL NAMES

C.1 RESERVED NAMES

ALL
ARRAY
ASSERT

BASED
BEGIN
BUFFER
BY

CALL
CASE
CAUSE
CONTINUE

DCL
DELAY
DO
DOWN

ELSE
ELSIF
END
ENTRY
ESAC
EVENT
EVER
EXCEPTIONS
EXIT

FI
FOR
FORBID

GENERAL
GOTO
GRANT

IF
IN
INIT
INLINE
INOUT

LOC

MODULE

NEWMODE
NOPACK

OD
OF
ON
OUT

PACK
PERVASIVE
POS
POWERSET
PRIORITY
PROC
PROCESS

RANGE
READ
RECEIVE
RECURSIVE
REF
REGION
RESULT
RETURN
RETURNS
ROW

SEIZE
SEND
SET
SIGNAL
SIMPLE
START
STATIC
STEP
STOP
STRUCT
SYN
SYNMODE

THEN
TO

UP

WHILE
WITH

C.2 PREDEFINED NAMES

ABS	FALSE	NOT	SIZE
ADDR		NULL	SUCC
AND		NUM	
	GETSTACK		
			THIS
BIN		OR	TRUE
BIT	INSTANCE		
BOOL	INT		
		PRED	UPPER
		PTR	
CARD	MAX		
CHAR	MIN		XOR
	MOD	REM	

C.3 CHILL EXCEPTION NAMES

ASSERTFAIL
DELAYFAIL
EMPTY
EXTINCT
MODEFAIL
OVERFLOW
RANGEFAIL
RECURSEFAIL
SPACEFAIL
TAGFAIL

C.4 CHILL DIRECTIVES

FREE

APPENDIX D: PROGRAM EXAMPLES

1. Operations on integers

```
integer_operations:
MODULE
   add:
   PROC (i,j INT)(INT) EXCEPTIONS (OVERFLOW);
      RESULT i+j;
   END add;
   mult:
   PROC (i,j INT)(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;
```

2. Same operations on fractions

```
fraction_operations:
MODULE
   NEWMODE fraction=STRUCT (num,denum INT);
   add:
   PROC (f1,f2 fraction)(fraction) EXCEPTIONS (OVERFLOW);
      RETURN [f1.num*f2.denum+f2.num*f1.denum,
          f1.denum*f2.denum];
   END add;
   mult:
   PROC (f1,f2 fraction)(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;
```

3. Same operations on complex numbers

```
1   complex_operations
2   MODULE
3      NEWMODE complex=STRUCT (re,im INT);
4      add:
5      PROC (c1,c2 complex)(complex) EXCEPTIONS (OVERFLOW);
6         RETURN [c1.re+c2.re,c1.im+c2.im];
7      END add;
8      mult:
9      PROC (c1,c2 complex)(complex) EXCEPTIONS (OVERFLOW);
10        RETURN [c1.re*c2.re-c1.im*c2.im ,
11                c1.re*c2.im+c1.im*c2.re];
12     END mult;
13
14     GRANT add, mult
15     SYNMODE operand_mode=complex;
16     GRANT operand_mode;
17     SYN neutral_for_add=complex [0,0],
18        neutral_for_mult=complex [1,0];
19     GRANT neutral_for_add,
20           neutral_for_mult;
21
22  END complex_operations:
```

4. General order arithmetic

```
1   general_order_arithmetic: /*from collected algorithms from CACM no.93*/
2   MODULE
3      op:
4      PROC (a INOUT, b,c,order INT) EXCEPTIONS (wrong_input) RECURSIVE;
5         DCL d INT;
6         ASSERT b>0 AND c>0 AND order>0
7            ON (ASSERTFAIL):
8               CAUSE wrong_input;
9            END;
10        CASE order OF
11           (1): a :=b+c;
12              RETURN;
13           (2): d :=0;
14           (ELSE): d :=1;
15        ESAC;
16        DO FOR i :=1 TO c;
17           op (a,b,d,order-1);
18           d :=a;
19        OD;
20        RETURN;
21     END op;
22
23     GRANT op;
24
25  END general_order_arithmetic;
```

5. Adding bit by bit and checking the result

```
1   add_bit_by_bit:
2   MODULE
3         adder:
4         PROC (a STRUCT(a2,a1 BOOL) IN, b STRUCT(b2,b1 BOOL) IN)
5             RETURNS(STRUCT(c4,c2,c1 BOOL));
6
7         DCL c STRUCT (c4,c2,c1 BOOL);
8         DCL k2,x,w,t,s,r BOOL;
9         DO WITH a,b,c;
10           k2 :=a1 AND b1;
11           c1 :=NOT k2 AND (a1 OR b1);
12           x :=e2 AND b2 AND k2;
13           w :=a2 OR b2 OR k2;
14           t :=b2 AND k2;
15           s :=a2 AND k2;
16           r :=a2 AND b2;
17           c4 :=r OR s OR t;
18           c2 :=x OR (w AND NOT c4);
19        OD;
20        RETURN c;
21     END adder;
22     GRANT adder;
23  END add_bit_by_bit;
24
25  exhaustive_checker:
26  MODULE
27     SEIZE adder;
28     DCL a STRUCT (a2,a1 BOOL),
29        b STRUCT (b2,b1 BOOL),
30     SYNMODE res=ARRAY (1:16) STRUCT (c4,c2,c1 BOOL);
31     DCL r INT, results res;
32     DO WITH a,b;
33        r :=0;
34        DO FOR a2 IN BOOL;
35           DO FOR a1 IN BOOL;
36              DO FOR b2 IN BOOL;
37                 DO FOR b1 IN BOOL;
38                    r+ :=1;
39                    results (r) :=adder (a,b);
40                 OD;
41              OD;
42           OD;
43        OD;
44        ASSERT result=res [[FALSE,FALSE,FALSE],[FALSE,FALSE,TRUE],
45                           [FALSE,FALSE,TRUE],[FALSE,TRUE,TRUE],
46                           [FALSE,FALSE,TRUE],[FALSE,TRUE,FALSE],
47                           [FALSE,TRUE,TRUE],[TRUE,FALSE,FALSE],
48                           [FALSE,TRUE,FALSE],[FALSE,TRUE,FALSE],
49                           [TRUE,FALSE,FALSE],[TRUE,FALSE,TRUE.],
50                           [FALSE,TRUE,TRUE],[TRUE,FALSE,FALSE],
51                           [TRUE,FALSE,TRUE],[TRUE,TRUE,FALSE]];
52  END exhaustive_checker;
```

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)(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;
      d :=4 * d + 3 - 1_461 * j;
      d :=(d + 4) / 4;
      m :=(5 * d - 3) / 153;
      d :=5 * d - 3 - 153 * m;
      d :=(d + 5) / 5;
      y :=100 * y + j;
      IF m<100 THEN m+:=3;
              ELSE m-:=9;
                   y+:=1;
      FI;
      RETURN [d,month (m+1), y];
   END gregorian_date;

   julian_day_number
   PROC (d date)(INT);
      DCL c,y,m INT;
      DO WITH d;
         m :=NUM (mo)+1;
         IF m>2 THEN m-:=3;
               ELSE m+:=9;
                    year-:=1;
         FI;
         c :=year/100;
         y :=year-100*c;
         RETURN (146_097*c)/4+(1_461*y)/4
            +(153+m+c)/5+day+1_721_119;
      OD;
   END julian_day_number;
   GRANT gregorian_date, julian_day_number;
END playing_with_dates

test:
MODULE
  SEIZE gregorian_date, julian_day_number;
  ASSERT julian_day_number ([10,dec,1979])=julian_day_number(
       gregorian_date(julian_day_number([10,dec,1979])));
END test;
```

7. Roman numerals

```
1   Roman:
2   MODULE
3      SEIZE n,rn;
4      convert:
5      PROC () EXCEPTIONS (string_too_small);
6         DCL r INT :=0;
7         DO WHILE n>=1_000;
8            rn(r):='M';
9            r+:=1;
10           n-:=1_000;
11        OD;
12        IF n>500 THEN rn(r):='D';
13                      n-:=500;
14                      r+:=1;
15        FI;
16        IF n>=100 THEN rn(r):='C';
17                       n-:=100;
18                       r+:=1;
19        FI;
20        IF n>=50 THEN rn(r):='L';
21                      n-:=50;
22                      r+:=1;
23        FI;
24        IF n>=10 THEN rn(r):='X';
25                      n-:=10;
26                      r+:=1;
27        FI;
28        DO WHILE n>=1;
29           rn(r):='I';
30           r+:=1;
31           n-:=1;
32        OD;
33        RETURN;
34     END ON (RANGEFAIL): DO FOR i :=0 TO UPPER (rn);
35                            rn(i) :='.';
36                         OD;
37                         CAUSE string_too_small;
39     END convert;
40  END Roman;
41  test:
42  MODULE
43     SEIZE convert;
44     DCL n INT INIT :=1979;
45     DCL rn CHAR (20) INIT :=(20)' ';
46     GRANT n,rn;
47
48     convert ();
49
50     ASSERT rn='MDCCCCLXXVIIII'//(6)' ';
51
52  END test;
```

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 CHAR (max) IN, output ARRAY('A':'Z') INT OUT);
7         DO FOR i :=0 TO UPPER (input ->);
8            IF input -> (i) IN letter
9               THEN
10                 output (input -> (i))+:=1;
11           FI;
12        OD;
13     END count;
14     GRANT count;
15  END letter-count;
16  test:
17  MODULE
18     DCL c CHAR (10) INIT :='A-B<ZAA9K''';
19     DCL output ARRAY ('A' : 'Z') INT;
20     SYN max=10_000;
21     GRANT max;
22     SEIZE count;
23     count (-> c,output);
24     ASSERT output=[('A') : 3,('B','K','Z') : 1, (ELSE) : 0];
25
26  END test;
```

9. Prime numbers

```
1   prime:
2   MODULE
3      SEIZE max;
4      NEWMODE number_list =POWERSET INT(2:max);
5      SYN  empty = number_list [];
6      DCL sieve number_list INIT := [2:max];
7         primes number_list INIT :=empty;
8      GRANT primes;
9      DO WHILE sieve/=empty;
10        primes OR :=[MIN (sieve)];
11        DO FOR j :=MIN (sieve) BY MIN (sieve) TO max;
12           sieve-:=[j];
13        OD;
14     OD;
15  END prime;
```

10. Implementing stacks in two different ways, transparent to the user

```
1   stacks_1:
2   MODULE
3      SEIZE element
4      SYN max=10_000,min=1;
5      DCL stack ARRAY (min : max) element,
6         stackindex INT INIT :=min;
7      push:
8      PROC (e element) EXCEPTIONS (overflow);
9         IF stackindex=max
10           THEN CAUSE overflow;
11        FI;
12        stackindex+:=1;
13        stack (stackindex) :=e;
14        RETURN;
15     END push;
16     pop:
17     PROC () EXCEPTIONS (underflow);
18        IF stackindex=min
19           THEN CAUSE underflow;
20        FI;
21        stackindex-:=1;
22        RETURN;
23     END pop;
24
25     elem:
26     PROC (i INT)(element LOC) EXCEPTIONS (bounds);
27        IF i<min OR i>max
28           THEN CAUSE bounds;
29        FI;
30        RETURN stack (i);
31     END elem;
32
33     GRANT push,pop,elem;
34  END stacks-1;
```

```
35  stacks_2:
36  MODULE
37     SEIZE element;
38     NEWMODE cell=STRUCT (pred,succ REF cell,
39                          info element);
40     DCL p,last,first REF cell INIT :=NULL;
41     push:
42     PROC (e element) EXCEPTIONS (overflow);
43        p :=allocate (cell) ON
44                            (nospace) : CAUSE overflow;
45                            END;
46        IF last=NULL
47           THEN first,last :=p;
48           ELSE last ->. succ :=p;
49              p ->. pred :=last;
50              last :=p;
51        FI;
52        last ->. info :=e;
53        RETURN;
54     END push;
55     pop:
56     PROC () EXCEPTIONS (underflow);
57        IF last=NULL;
58           THEN CAUSE underflow;
59        FI;
60        last :=last ->. pred; IF last = NULL THEN first := NULL FI;
61        last ->. succ :=NULL;
62        RETURN;
63     END pop;
64     elem:
65     PROC (i INT) (element LOC) EXCEPTIONS (bounds);
66        IF first=NULL
67           THEN CAUSE bounds;
68        FI;
69        p :=first;
70        DO FOR j=2 TO i;
71           IF p ->. succ=NULL
72              THEN CAUSE bounds;
73           FI;
74           p :=p ->. succ;
75        OD;
76        RETURN p ->. info;
77     END elem;
78
79     GRANT push,pop,elem;
80
81  END stacks_2;
```

11. Fragment for playing chess

```
1   NEWMODE piece=STRUCT(color SET(white,black),
2                        kind SET(pawn,rook,knight,bishop,queen,king));
3   NEWMODE column=SET (a,b,c,d,e,f,g,h);
4   NEWMODE line=INT (1 : 8);
5   NEWMODE square=STRUCT (status SET (occupied,free),
6                          CASE status OF
7                          (occupied) : p piece,
8                          (free) :
9                          ESAC);
10  NEWMODE board=ARRAY (line) ARRAY (column) square;
11  NEWMODE move=STRUCT (lin_1,lin_2 line,
12                       col_1,col_2 column);
13
14  initialise:
15  PROC (bd board INOUT);
16     bd :=[(1) : [(a,h):[.status: occupied, .p : [white,rook]],
17           (b,g):[.status: occupied, .p : [white,knight]],
18           (c,f):[.status: occupied, .p : [white,bishop]],
19             (d):[.status: occupied, .p : [white,queen]],
20             (e):[.status: occupied, .p : [white,king]]],
21    (2):[(ELSE) : [.status: occupied, .p : [white,pawn]]],
22   (3:6):[(ELSE) : [.status: free]],
23     (7):[(ELSE) : [.status: occupied, .p : [black,pawn]]],
24     (8):[(a,h) : [.status: occupied, .p : [black,rook]],
25       (b,g):[.status: occupied, .p : [black,knight]],
26             (e,f) : [.status: occupied, .p : [black,bishop]],
27               (d) : [.status: occupied, .p : [black,king]],
28               (e) : [.status: occupied, .p : [black,queen]]]];
29     RETURN;
30
31  END initialise;
```

```
register_move:
PROC (b board LOC,m move) EXCEPTIONS (illegal);
   DCL starting square LOC :=b (m.lin_1)(m.col_2),
      arriving square LOC :=b (m.lin_1)(m.col_2);

   DO WITH m;
      IF starting.status=free
         OR (lin_2<1 OR lin_2>8 OR col_2<a OR col_2>h)
         OR (arriving.status/=free AND arriving.p.kind=king)
         THEN
            CAUSE illegal;
      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=SUCC(col_1))
             AND arriving.status=free OR arriving.p.color=white
          THEN
              CAUSE illegal; /*capturing en passant not implemented*/
          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=SUCC(col_1))
             AND arriving.status=free OR arriving.p.color=black
             THEN
                CAUSE illegal; /* same remark */
          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)
             THEN
                IF NOT ok_bishop (b,m)
                   THEN
                      CAUSE illegal
                FI;
         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
            OR arriving.status/=free AND
```

```
86                arriving.p.color=starting.p.color
87            THEN CAUSE illegal;
88            FI;
89              (king),(*):
90               IF ABS(NUM(col_2)-NUM(col_1)) > 1
91                  OR ABS(lin_2-lin_1) > 1
92                  OR lin_2=lin_1 AND col_2=col_1
93                  OR arriving.status/=free AND
94                     arriving.p.color=starting.p.color
95               THEN CAUSE illegal;
96               FI; /*checking king moving to check not implemented*/
97           ESAC;
98       OD;
99       arriving :=starting;
100      RETURN;
101   END register_move;
102   ok_rook:
103   PROC (b board,m move)(BOOL);
104      DO WITH m;
105         IF NOT (col_2=col_1 OR lin_1=lin_2)
106            OR arriving.status/=free AND
107            arriving.p.color=starting.p.color
108         THEN RETURN FALSE;
109         FI;
110         IF col_1=col_2
111            THEN IF lin_1<lin_2
112                    THEN DO FOR l := lin_1+1 TO lin_2-1;
113                            IF board (l)(col_1).status/=free
114                               THEN RETURN FALSE;
115                            FI;
116                         OD;
117                    ELSE DO FOR l := lin_1-1 DOWN TO lin_2+1;
118                            IF board (l)(col_1).status/=free
119                               THEN RETURN FALSE;
120                            FI;
121                         OD;
122                 FI;
123            ELSE IF col_1<col_2
124                    THEN DO FOR c := SUCC(col_1) TO PRED(col_2);
125                            IF board (lin_1)(c).status/=free
126                               THEN RETURN FALSE;
127                            FI;
128                         OD;
129                    ELSE DO FOR c := SUCC(col_2) DOWN TO PRED(col_1);
130                            IF board (lin_1)(c).status/=free
131                               THEN RETURN FALSE;
132                            FI;
133                         OD;
134                 FI;
135         FI;
136         RETURN TRUE;
137      OD;
138   END ok_rook;
```

```
139  ok_bishop:
140  PROC (b board,m move)(BOOL);
141     DO WITH m;
142        CASE lin_2>lin_1,col_2>col_1 OF
143           (TRUE),(TRUE): c := col_1
144                       DO FOR  l := lin_1+1 TO lin_2-1;
145                          c := SUCC(c);
146                          IF board (l)(c).status/=free
147                             THEN RETURN FALSE;
148                          FI;
149                       OD;
150                       IF SUCC(c)/=col_2
151                          THEN RETURN FALSE;
152                       FI;
153           (TRUE),(FALSE): c := col_1
154                       DO FOR l := lin_1+1 TO lin_2-1;
155                          c := PRED(c);
156                          IF board (l)(c).status/=free
157                             THE RETURN FALSE;
158                          FI;
159                       OD;
160                       IF PRED(c)/=col_2
161                          THEN RETURN FALSE;
162                       FI;
163           (FALSE),(TRUE): c := col_1
164                       DO FOR l := lin_1-1 DOWN TO lin_2+1;
165                          c := SUCC(c)
166                          IF board (l)(c).status/=free
167                             THEN RETURN FALSE;
168                          FI;
169                       OD;
170                       IF SUCC(c)/=col_2
171                          THEN RETURN FALSE;
172                       FI:
173           (FALSE),(FALSE): c := col_1;
174                        DO FOR l := lin_1-1 DOWN TO lin_2+1;
175                           c := PRED(c);
176                           IF board (l)(c).status/=free
177                              THEN RETURN FALSE;
178                           FI;
179                        OD;
180                        IF PRED (c)/=col_2
181                           THEN RETURN FALSE;
182                        FI;
183        ESAC;
184        RETURN arriving.status=free OR
184              arriving.p.color/=starting.p.color;
186     OD;
187  END ok_bishop;
```

12. Building and manipulating a circularly linked list

```
1   CIRCULAR_LIST:
2   MODULE

3       HANDLE_LIST:
4       MODULE
5           GRANT INSERT, REMOVE, NODE;
6           NEWMODE NODE=STRUCT(PRED, SUC REF NODE, VALUE INT);
7           DCL POOL ARRAY(1:1000)NODE;
8           DCL HEAD NODE:=(: NULL,NULL,0 :);
9           INSERT:
            PROC(NEW NODE);
10              /* INSERT ACTIONS */
11          END INSERT;

12          REMOVE:
            PROC();
13              /* REMOVE ACTIONS */
14          END REMOVE;

15          INITIALIZE_LIST:
16          BEGIN
17              DCL LAST REF NODE:= ->HEAD;
18              DO FOR NEW IN POOL;
19                  NEW.PRED := LAST;
20                  LAST->.SUC:= ->NEW;
21                  LAST:= ->NEW;
22                  NEW.VALUE:=0;
23              OD;
24              HEAD.PRED:=LAST;
25              LAST->.SUC:= ->HEAD;
26          END INITIALIZE_LIST

27      END HANDLE_LIST;

28      DCL NODE_A NODE:=(: NULL,NULL,536 :);
29      REMOVE();
30      REMOVE();
31      INSERT(NODE_A);
32  END CIRCULAR_LIST;
```

13. **A region for managing competing accesses to a resource**

```
1   ALLOCATE_RESOURCES:
2   REGION
3       GRANT ALLOCATE, DEALLOCATE;
4       NEWMODE RESOURCE_SET = INT(0:9);
5       DCL ALLOCATED ARRAY(RESOURCE_SET)BOOL :=
                    (: (RESOURCE_SET): FALSE :);
6       DCL RESOURCE_FREED EVENT;

7       ALLOCATE:
8       PROC()(INT);
9           DO FOR EVER;
10              DO FOR I IN RESOURCE_SET;
11                  IF NOT ALLOCATED(I)
12                  THEN
13                      ALLOCATED(I) := TRUE;
14                      RETURN I;
15                  FI;
16              OD;
17              DELAY RESOURCE_FREED;
18          OD;
19      END ALLOCATE;

20      DEALLOCATE:
21      PROC(I INT);
22          ALLOCATE(I) := FALSE;
23          CONTINUE RESOURCE_FREED;
24      END DEALLOCATE;

25  END ALLOCATE_RESOURCES;
```

14. Queuing calls to a switchboard

```
1   SWITCHBOARD:
2   MODULE
3    /* This example illustrates a switchboard which queues incoming calls
4       and feeds them to the operator at an even rate. Every time the
5       operator is ready one and only one call is let through. This is
6       handled by a call distributor which lets calla through at fixed
7       intervals. If the operator is not ready or there are other calls
8       waiting, a new call must queue up to wait for its turn. */

9       DCL OPERATOR_IS_READY,
10          SWITCH_IS_CLOSED EVENT;

11      CALL_DISTRIBUTOR:
12      PROCESS();
13          DO FOR EVER;
14              WAIT(10 /*seconds*/);
15              CONTINUE OPERATOR_IS_READY;
16          OD;
17      END CALL_DISTRIBUTOR;

18      CALL:
19      PROCESS();
20          DELAY CASE
21          (OPERATOR_IS_READY): /* some actions */
22          (SWITCH_IS_CLOSED): DO FOR I IN INT(1:100);
23                                  CONTINUE OPERATOR_IS_READY;
24                                  /*empty the queue*/
25                              OD;
26          ESAC;
27      END CALL;

28      OPERATOR:
29      PROCESS();
30          DO FOR_EVER;
31              IF TIME = 1700
32                  THEN
33                      CONTINUE SWITCH_IS_CLOSED;
34              FI;
35          OD;
36      END OPERATOR;

37      START CALL_DISTRIBUTOR():
38      START OPERATOR();
39      DO FOR I INT(1:100);
40          START CALL();
41      OD;
42  END SWITCHBOARD;
```

15. Allocating and deallocating a set of resources

```
1   <> FREE (STEP);
2   COUNTER MANAGER:
3   MODULE
4   /* To illustrate the use of signals and the receive case, (buffers
5      might have been instead) we will look at an example where an
6      ALLOCATOR manages a set of resources, in this case a set of
7      COUNTERs. The module is part of a larger system where there are
8      USERs, that can request the services of the COUNTER_MANAGER. The
9      module is made to consistof two process definitions, one for the
10     ALLOCATION and one for the COUNTERS. INITIATE and TERMINATE
11     are internal signals sent from the ALLOCATOR
12     to the COUNTERs. All the other signals are external, being sent
13     from or to the USERs. */

14      SEIZE /* external signals */
15         ACQUIRE, RELEASE, CONGESTED,STEP,READOUT;
16      SIGNAL INITIATE = (INSTANCE),
17             TERMINATE;

18      ALLOCATOR:
19      PROCESS();
20         NEWMODE NO_OF_COUNTERS = INT(1:100);
21      DCL COUNTERS ARRAY (NO_OF_COUNTERS)
22                 STRUCT (COUNTER INSTANCE,
23                         STATUS SET (BUSY,IDLE));
24      DO FOR EACH IN COUNTERS;
25      EACH:= (: START COUNTER(), IDLE :);
26      OD;

27      DO FOR EVER;
28      BEGIN
29         DCL USER INSTANCE;
30         AWAIT_SIGNALS:
31         RECEIVE CASE SET USER;
32         (ACQUIRE):
33            DO FOR EACH IN COUNTERS;
34               DO WITH EACH;
35                  IF STATUS = IDLE
36                  THEN
37                     STATUS:=BUSY;
38                     SEND INITIATE (USER) TO COUNTER;
39                     EXIT AWAIT_SIGNALS;
40                  FI;
41               OD;
42            OD;
43            SEND CONGESTED TO USER;
44         (RELEASE IN THIS_COUNTER);
45            SEND TERMINATE TO THIS_COUNTER;
```

```
46              FIND_COUNTER:
47              DO FOR EACH IN COUNTERS;
48                 DO WITH EACH;
49                    IF THIS_COUNTER = COUNTER
50                    THEN
51                       STATUS:= IDLE;
52                       EXIT FIND_COUNTER;
53                    FI;
54                 OD;
55              OD FIND_COUNTER;
56           ESAC AWAIT_SIGNALS;
57        END;
58        OD;
59        END ALLOCATOR;

60        COUNTER:
61        PROCESS();
62           DO FOR EVER;
63           BEGIN
64              DCL USER INSTANCE;
65                 COUNT:= 0;
66              RECEIVE CASE
67                 (INITIATE IN RECEIVED_USER):
68                    SEND READY TO RECEIVED_USER;
69                    USER:= RECEIVED_USER;
70              ESAC;
71              WORK_LOOP:
72              DO FOR EVER;
73                 RECEIVE CASE
74                    (STEP): COUNT +:=1;
75                    (TERMINATE):
76                       SEND READOUT(COUNT) TO USER;
77                       EXIT WORK_LOOP;
78                 ESAC;
79              OD WORK_LOOP;
80           END;
81           OD;
82        END COUNTER;

83        START ALLOCATOR();
84     END COUNTER_MANAGER;
```

16. Allocating and deallocating a set of resources using buffers

```
<> FREE(STEP);
USER_WORLD:
MODULE
/* This example is the same as no.15 except that buffers are
   used for communication in stead 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. */

GRANT USER_BUFFERS,
      ALLOCATOR_MESSAGES, ALLOCATOR_BUFFERS,
      COUNTER_MESSAGES, COUNTER_BUFFERS;
NEWMODE
  USER_MESSAGES =
      STRUCT(TYPE SET(CONGESTED, READY,
                      READOUT, ALLOCATOR_ID),
             CASE TYPE OF
             (CONGESTED)   :
             (READY)       : COUNTER REF COUNTER_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 COUNTER_BUFFERS
             ESAC),
  ALLOCATOR_BUFFERS = BUFFER(1) ALLOCATOR_MESSAGES,
  COUNTER_MESSAGES =
      STRUCT(TYPE SET(INITIATE, STEP, TERMINATE),
             CASE TYPE OF
             (INITIATE) : USER REF USER_BUFFERS,
             (STEP,
              TERMINATE):
             ESAC,
  COUNTER_BUFFERS = BUFFER(1) COUNTER_MESSAGES;
DCL USER_BUFFER USER_BUFFERS,
    ALLOCATOR_BUF REF ALLOCATOR_BUFFERS,
    COUNTER_BUF REF COUNTER_BUFFERS;
START ALLOCATOR(->USER_BUFFER);
ALLOCATOR_BUF := (RECEIVE USER_BUFFER).ALLOCATOR;
END_USER_WORLD;
```

```
51  COUNTER_MANAGER:
52  MODULE
53  SEIZE USER_BUFFERS,
54        ALLOCATOR_MESSAGES, ALLOCATOR_BUFFERS,
55        COUNTER_MESSAGES, COUNTER_BUFFERS;
56
57  ALLOCATOR:
58  PROCESS(STARTER REF USER_BUFFERS);
59     DCL ALLOCATOR_BUFFER ALLOCATOR_BUFFERS;
60     NEWMODE NO_OF_COUNTERS = INT(1:10);
61     DCL COUNTERS ARRAY(NO_OF_COUNTERS)
62           STRUCT(COUNTER REF COUNTER_BUFFERS,
63                  STATUS SET(BUSY, IDLE)),
64          MESSAGE ALLOCATOR_MESSAGES;
65     SEND STARTER->([ALLOCATOR_ID, ->ALLOCATOR_BUFFER]);
66     DO FOR EACH IN COUNTERS;
67        START COUNTER(->ALLOCATOR_BUFFER);
68        EACH := [(RECEIVE ALLOCATOR_BUFFER).COUNTER, IDLE];
69     OD;
70     DO FOR EVER;
71        BEGIN
72        DCL USER REF USER_BUFFERS;
73        MESSAGE := RECEIVE ALLOCATOR_BUFFER;
74        HANDLE_MESSAGES:
75        CASE MESSAGE.TYPE OF
76        (ACQUIRE):
77           USER := MESSAGE.USER;
78           DO FOR EACH IN COUNTERS;
79              DO WITH EACH;
80                 IF STATUS= IDLE
81                 THEN STATUS := BUSY;
82                      SEND COUNTER->([INITIATE, USER]);
83                      EXIT HANDLE_MESSAGES;
84                 FI;
85              OD;
86           OD;
87           SEND USER->([CONGESTED]);
88        (RELEASE):
89           SEND MESSAGE.COUNTER([TERMINATE]);
90           FIND_COUNTER:
91           DO FOR EACH IN COUNTERS;
92              DO WITH EACH;
93                 IF MESSAGE.COUNTER = COUNTER
94                 THEN STATUS := IDLE;
95                      EXIT FIND_COUNTER;
96                 FI;
97              OD;
98           OD FIND_COUNTER;
99        ESAC HANDLE_MESSAGES;
100       END;
101    OD;
102 END ALLOCATOR;
```

```
103 COUNTER:
104 PROCESS(STARTER REF ALLOCATOR_BUFFERS);
105    DCL COUNTER_BUFFER ALLOCATOR_BUFFERS;
106    SEND STARTER->([COUNTER_ID, ->COUNTER_BUFFER]);
107    DO FOR EVER;
108       BEGIN
109         DCL USER REF USER_BUFFERS,
110             COUNT INT := 0,
111             MESSAGE COUNTER_MESSAGES;
112         MESSAGE := RECEIVE COUNTER_BUFFER;
113         CASE MESSAGE.TYPE OF
114         (INITIATE): USER := MESSAGE.USER;
115                     SEND USER->([READY, ->COUNTER_BUFFER]);
116         ELSE /* some error action */
117         ESAC;
118         WORK_LOOP:
119         DO FOR EVER;
120            MESSAGE := RECEIVE COUNTER_BUFFER;
121            CASE MESSAGE.TYPE OF
122            (STEP)      : COUNT +:= 1;
123            (TERMINATE):SEND USER->([READOUT, COUNT]);
124                        EXIT WORK_LOOP;
125            ELSE /* some error action */
126            ESAC;
127         OD WORK_LOOP;
128    END;
129    OD;
130 END COUNTER;
131 END COUNTER_MANAGER;
```

17. String scanner1

```
1  string_scanner1: /* This program implements strings by means
2                      of packed arrays of characters.*/
3  MODULE
4  SYN
5    blanks ARRAY(0:9)CHAR PACK = [(*):' '], linelength = 132;
6  SYNMODE
7    stringptr = ROW ARRAY(lineindex)CHAR PACK,
8    lineindex = INT(0:linelength-1);
9
10 scanner:
11 PROC(string stringptr, scanstart lineindex INOUT,
12      scanstop lineindex, stopset POWERSET CHAR)
13     RETURNS(ARRAY(0:9)CHAR PACK);
14     DCL count INT:=0,
15         res ARRAY(0:9)CHAR PACK:=blanks;
16     DO
17       FOR c IN string->(scanstart:scanstop)
18       WHILE NOT (c IN stopset);
19         count+:=1;
20     OD;
21     IF count>0
22       THEN
23           IF count>10
24             THEN
25               count:=10;
26            FI;
27            res(0:count-1):=string->(scanstart:scanstart+count-1);
28     FI;
29     RESULT res;
30     IF scanstart+count < scanstop
31       THEN
32           scanstart:=scanstart+count+1;
33     FI;
34     END scanner;
35
36 GRANT
37    scanner;
38
39 END string_scanner;
```

18. String scanner2

```
1  string_scanner2: /* This example is the same as no.18 but it uses
2                      character string in stead of packed arrays.*/
3  MODULE
4    SYN
5      blanks = (10)' ', linelength = 132;
6    SYNMODE
7      stringptr = ROW CHAR(linelength),
8      lineindex = INT(0:linelength-1);
9
10   scanner:
11     PROC(string stringptr, scanstart lineindex INOUT,
12         scanstop lineindex, stopset POWERSET CHAR)
13       RETURNS (CHAR(10));
14       DCL count INT:=0;
15       DO FOR i := scanstart TO scanstop
16          WHILE NOT (string->(i) IN stopset);
17            count+:=1;
18       OD;
19       IF count>0
20         THEN
21           IF count>=10
22             THEN
23               RESULT string->(scanstart UP 10);
24             ELSE
25               RESULT string->(scanstart:scanstart+count-1)
26                     //blanks(count:9);
27           FI;
28         ELSE
29           RESULT blanks;
30       FI;
31       IF scanstart+count < scanstop
32         THEN
33           scanstart:=scanstart+count+1;
34       FI;
35     END scanner;
36
37   GRANT
38     scanner;
39
40 END string_scanner;
```

19. Removing an item from a double linked list

```
 1  QUEUE_REMOVAL:
 2  MODULE
 3     SEIZE INFO;
 4     GRANT REMOVE;
 5     REMOVE: PROC(P PTR) RETURNS(INFO) EXCEPTIONS(EMPTY);
 6     /* This procedure removes the item referred to by
 7        P from a queue and returns the information contents
 8        of that queue element.*/
 9          DCL 1 X BASED (P),
10                2 I INFO POS(0,8:31),
11                2 PREV PTR POS(1,0:15),
12                2 NEXT PTR POS(1,16:31);
13          DCL PREV, NEXT PTR;
14          PREV := X.PREV;
15          NEXT := X.NEXT;
16          X.PREV, X.NEXT := NULL;
17          RESULT X.INFO;
18          P := PREV;
19          X.NEXT := NEXT;
20          P := NEXT;
21          X.PREV := PREV;
22     END REMOVE;
23
24  END QUEUE_REMOVAL;
```

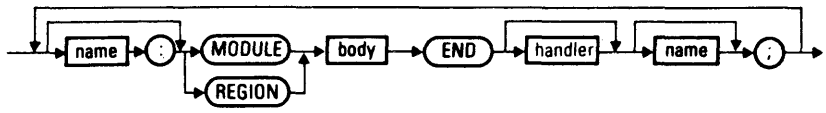
APPENDIX E: SYNTAX DIAGRAMS

The diagrams in this appendix describe the syntax of CHILL.

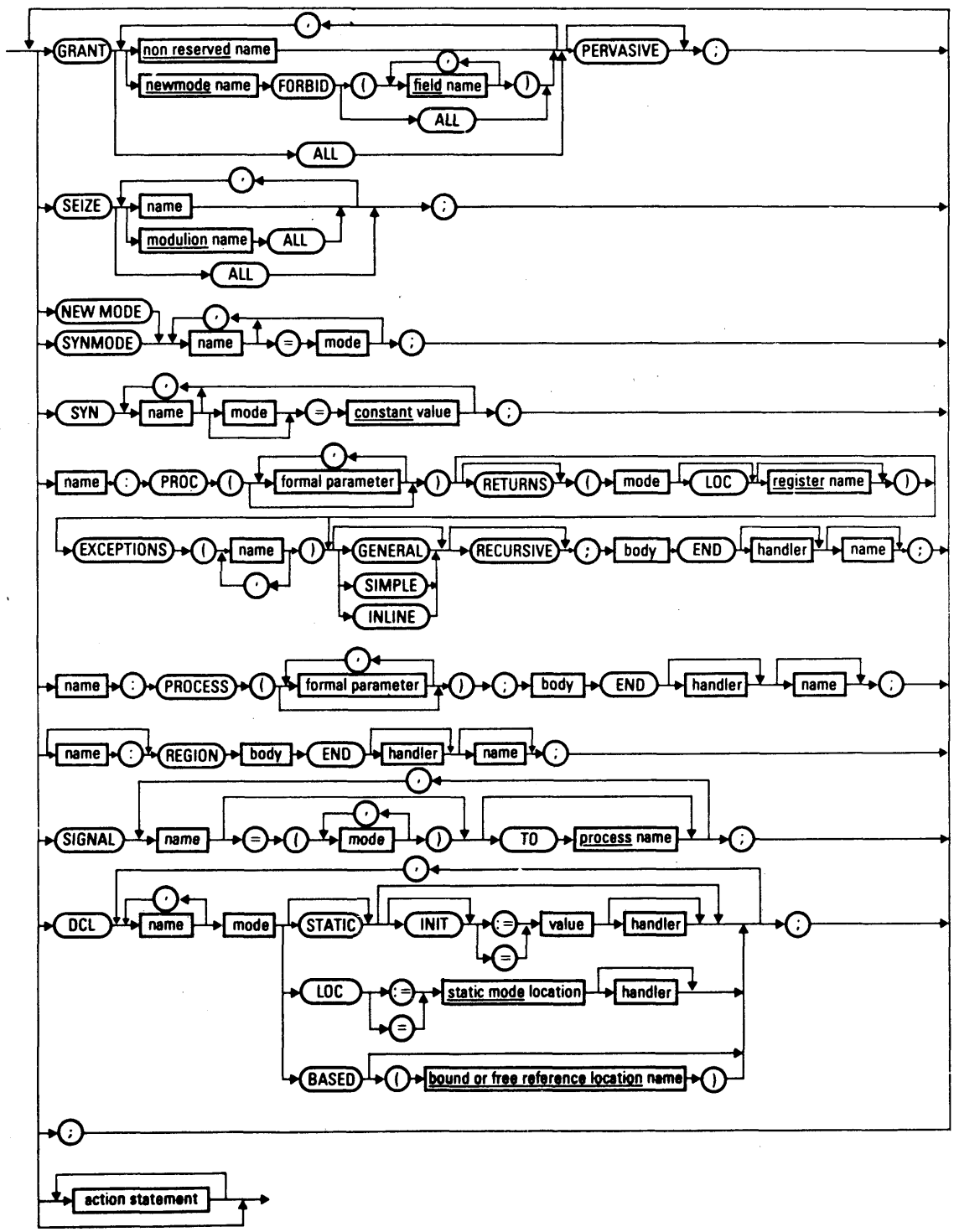
The diagrams have been designed for human readability, not as a basis for parsing.

Simplifications have been made in order to enhance readability and therefore they cannot be considered as definitive, only as an aid to understanding CHILL. The definition of the context-free syntax is specified in Backus-Naur Form elsewhere in this document.

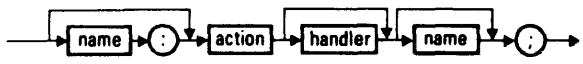
program



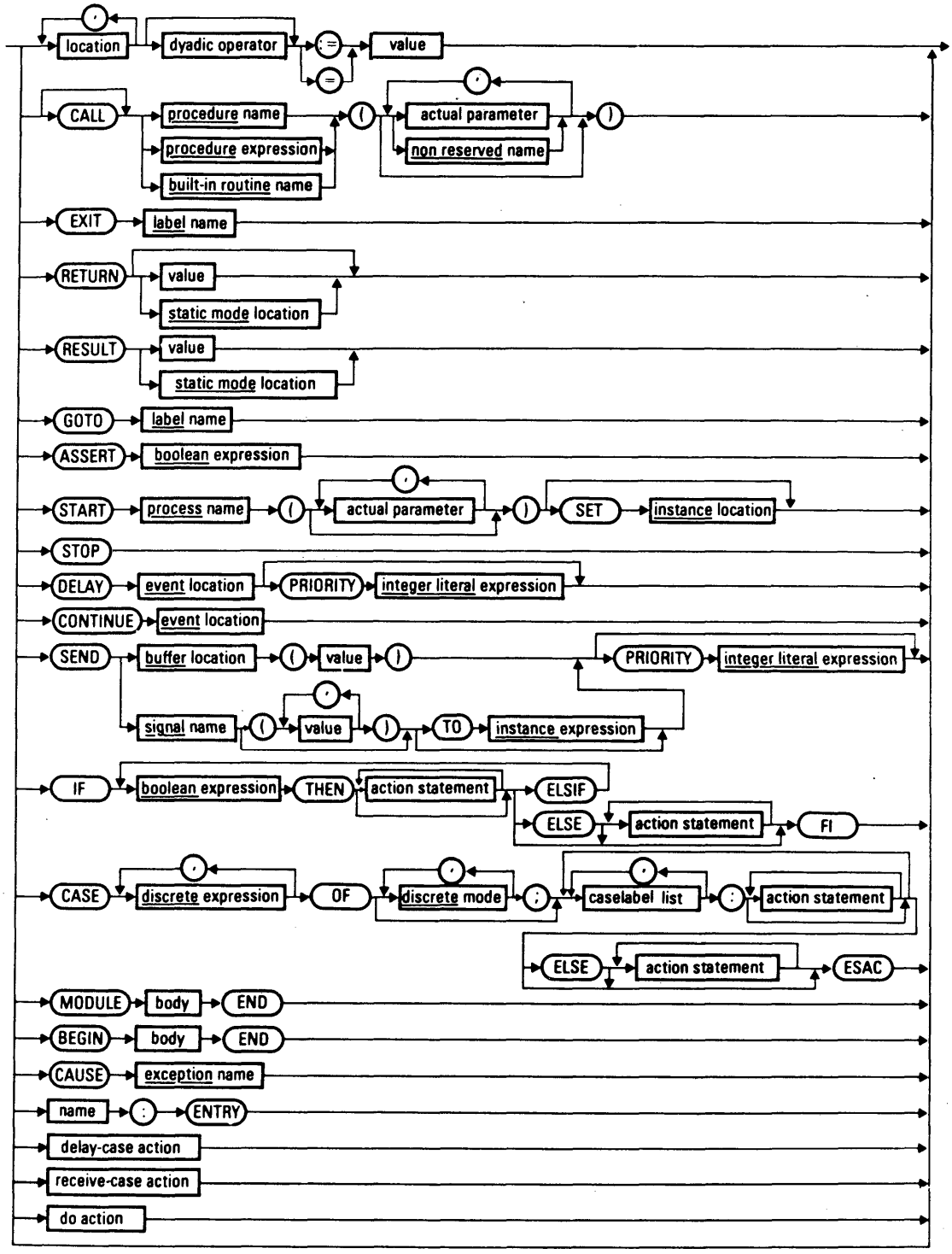
body



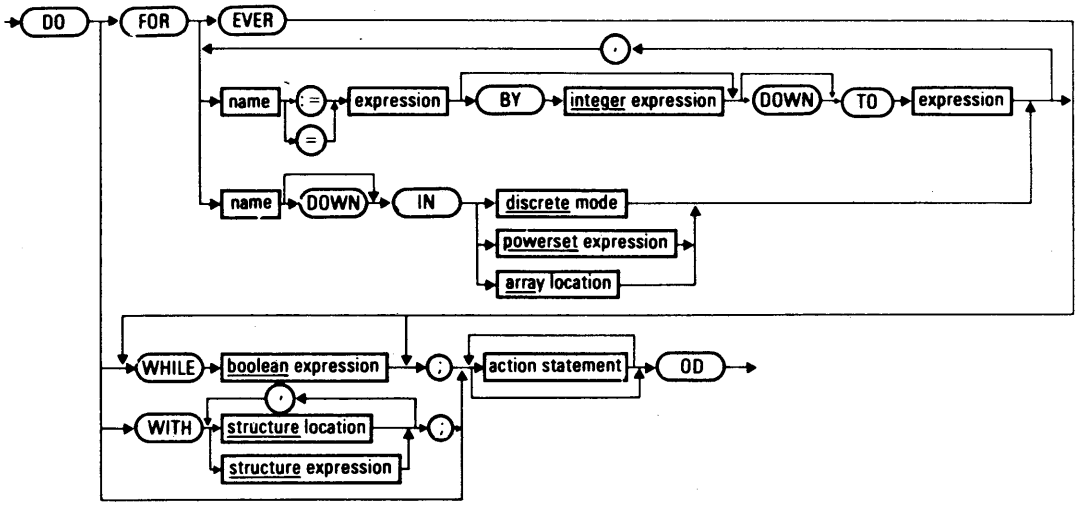
action statement



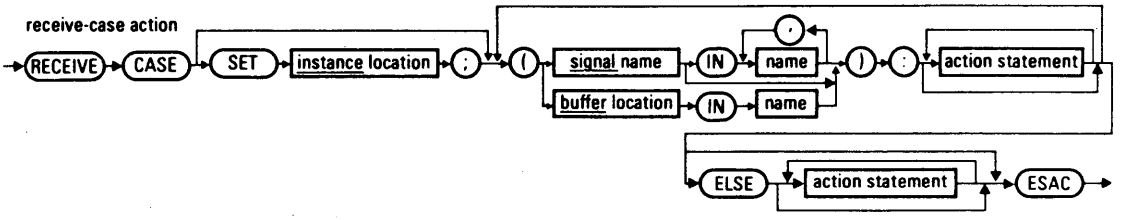
action



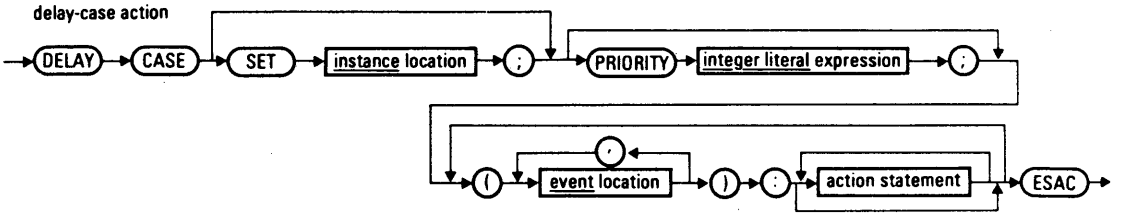
do action



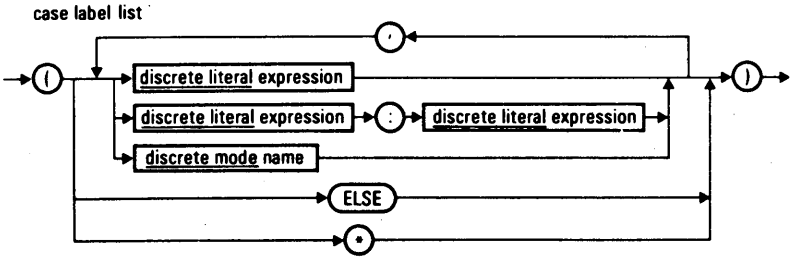
receive-case action



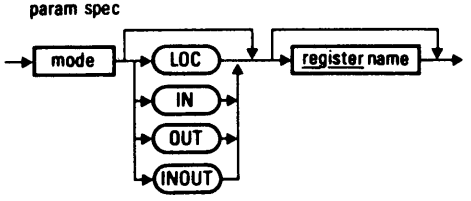
delay-case action



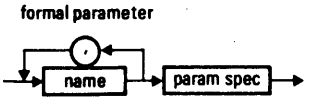
case label list



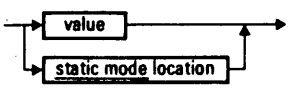
param spec

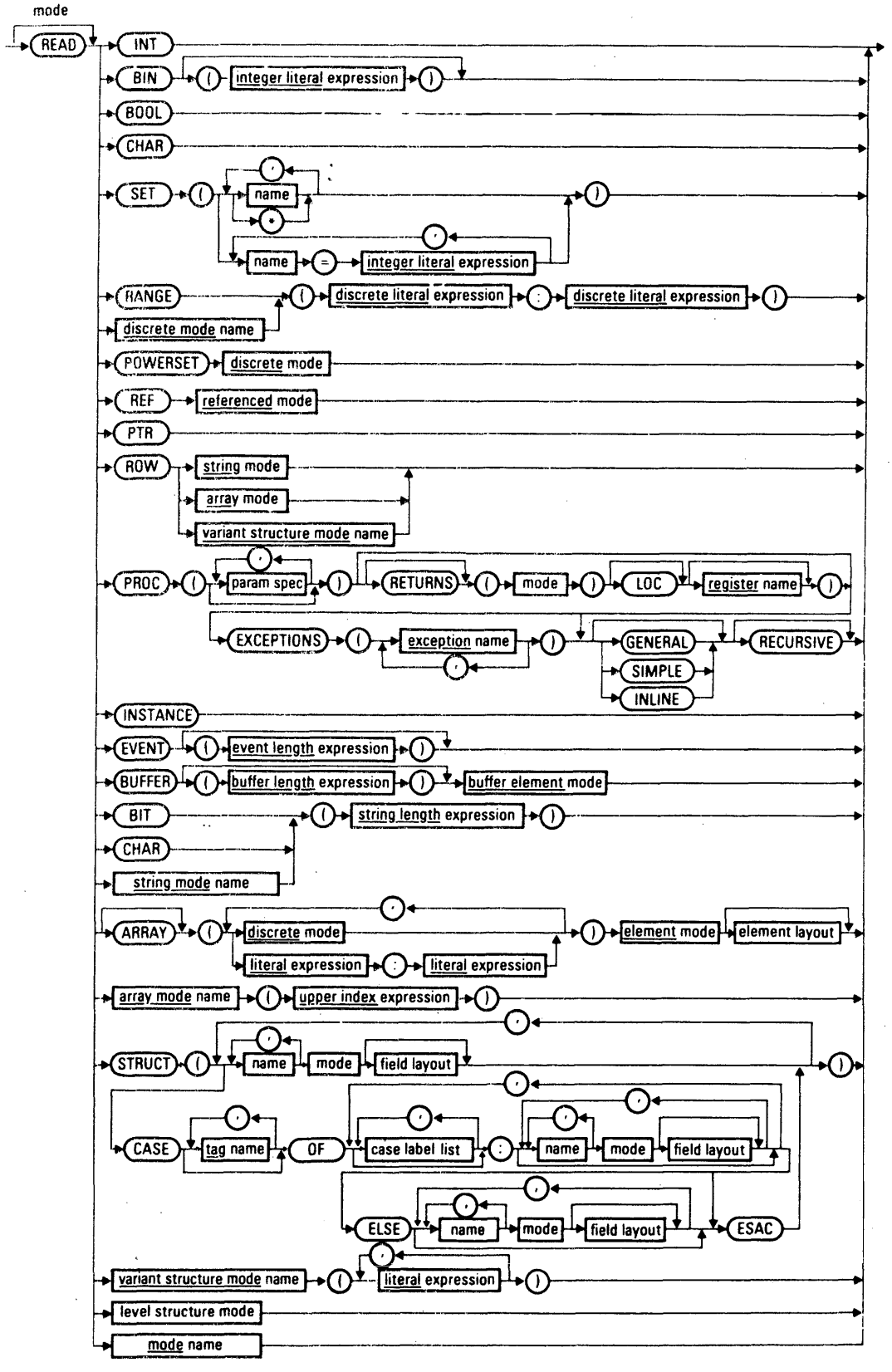


formal parameter



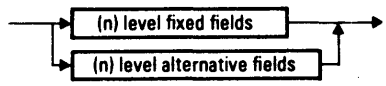
actual parameter



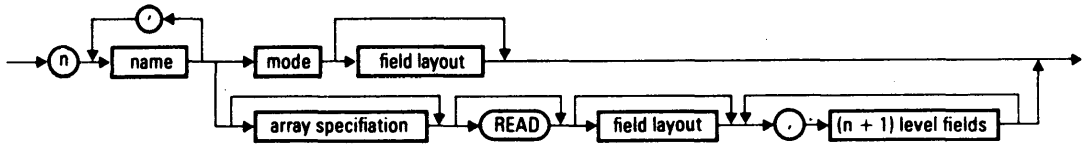

mode
READ
INT
BIN
integer literal expression
BOOL
CHAR
SET
name
integer literal expression
RANGE
discrete literal expression
discrete literal expression
discrete mode name
POWERSET
discrete mode
REF
referenced mode
PTR
ROW
string mode
array mode
variant structure mode name
PROC
param spec
RETURNS
mode
LOC
register name
EXCEPTIONS
exception name
GENERAL
SIMPLE
INLINE
RECURSIVE
INSTANCE
EVENT
event length expression
BUFFER
buffer length expression
buffer element mode
BIT
string length expression
CHAR
string mode name
ARRAY
discrete mode
literal expression
literal expression
element mode
element layout
array mode name
upper index expression
STRUCT
name
mode
field layout
CASE
tag name
OF
case label list
name
mode
field layout
ELSE
name
mode
field layout
ESAC
variant structure mode name
literal expression
level structure mode
mode name

level structure mode

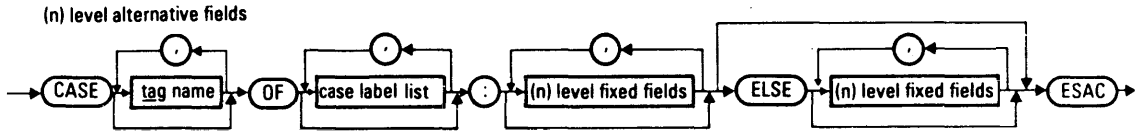
(n) level fields



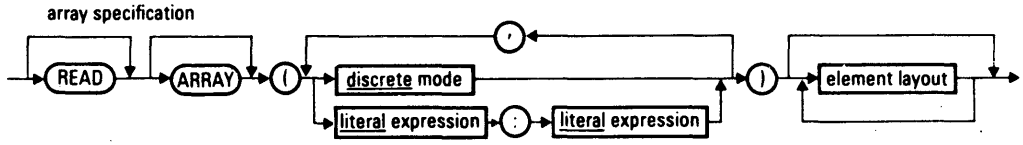
(n) level fixed fields



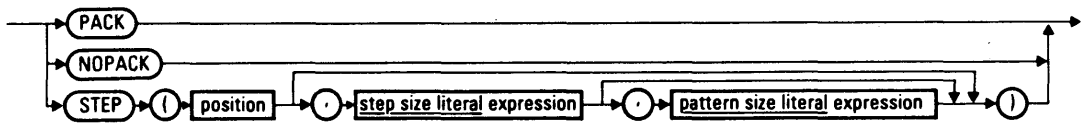
(n) level alternative fields



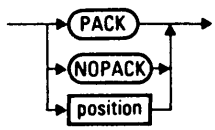
array specification



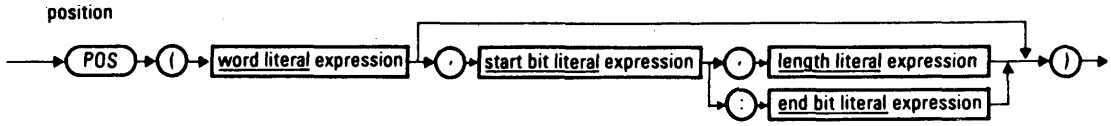
element layout

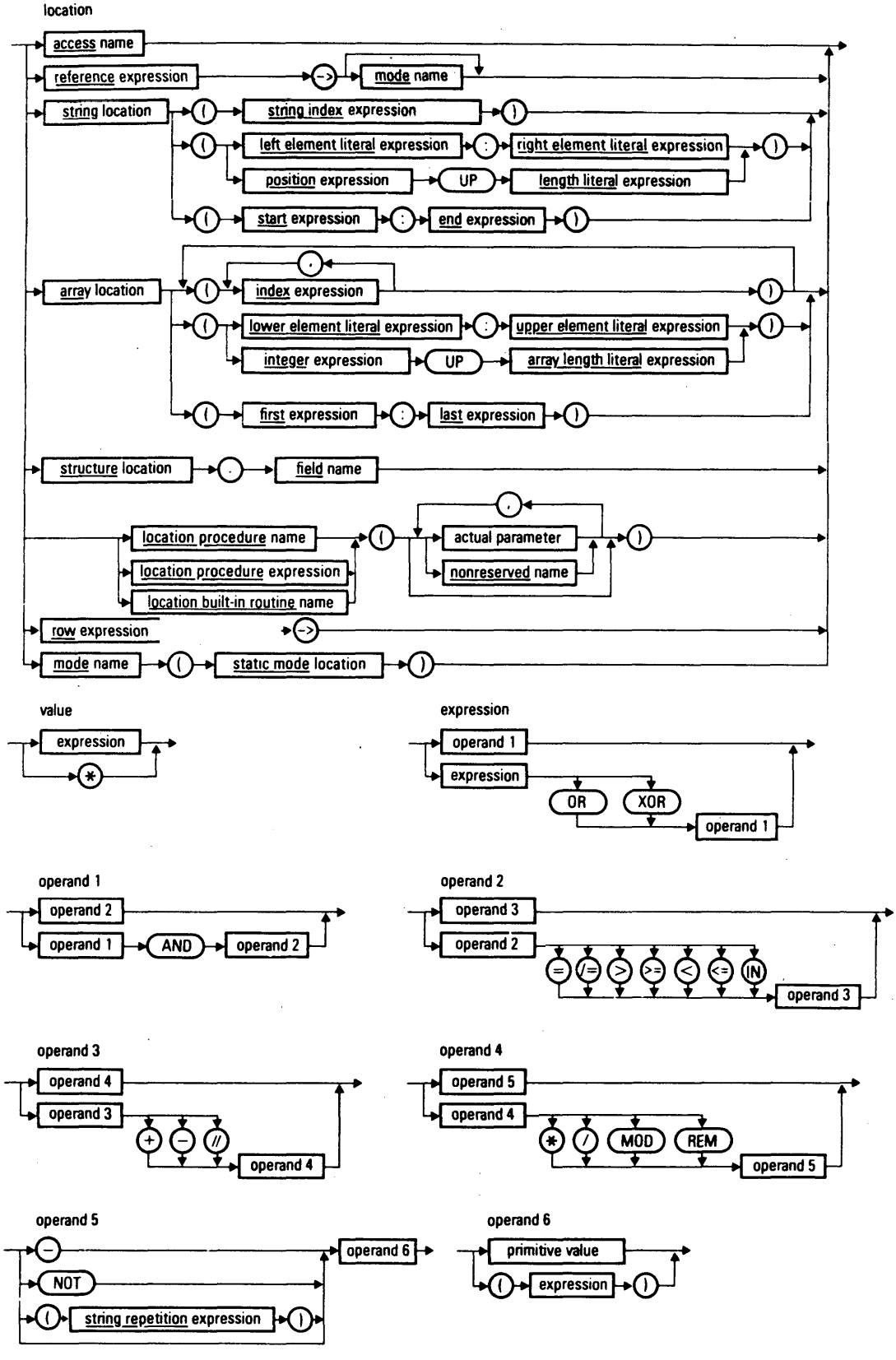


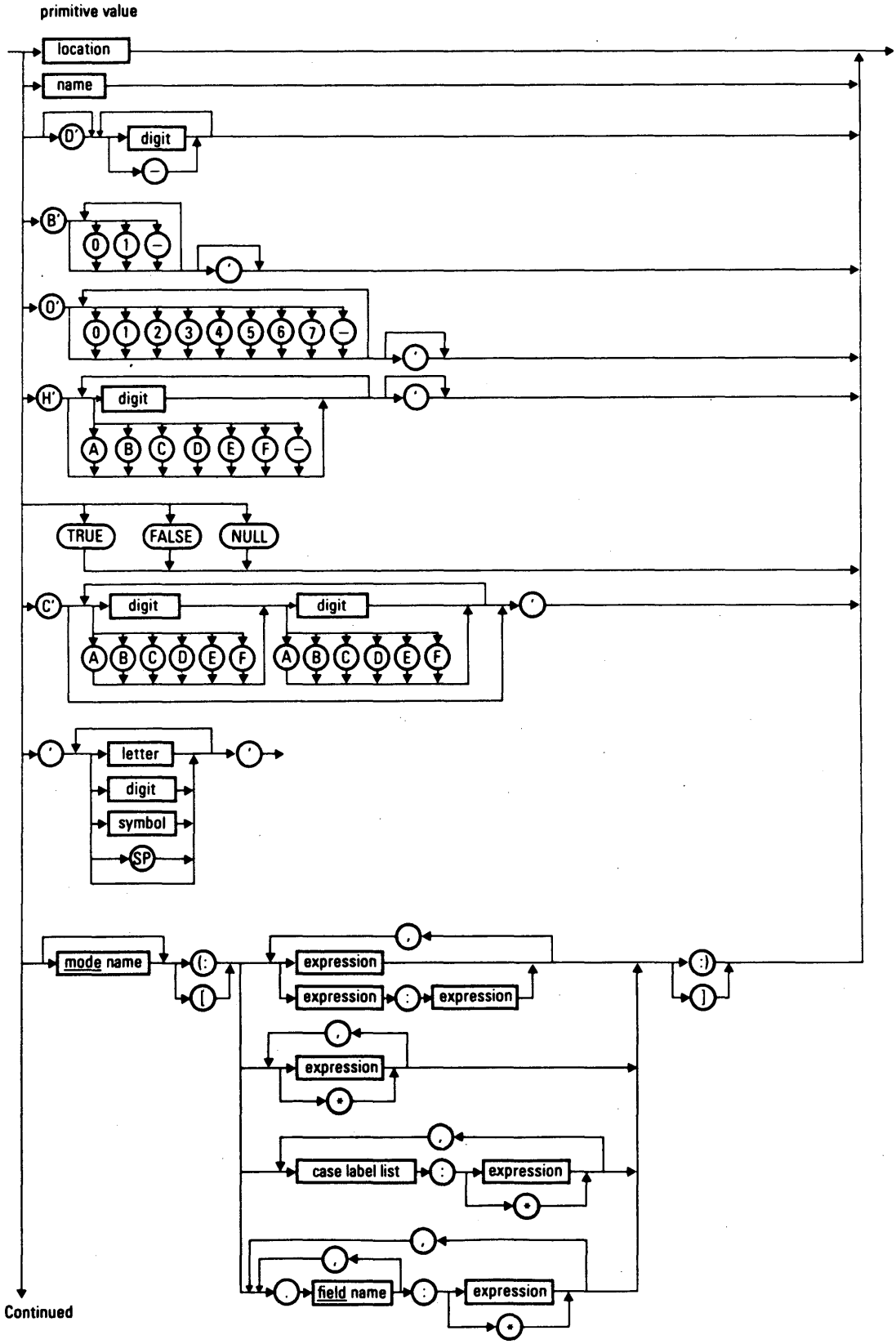
field layout



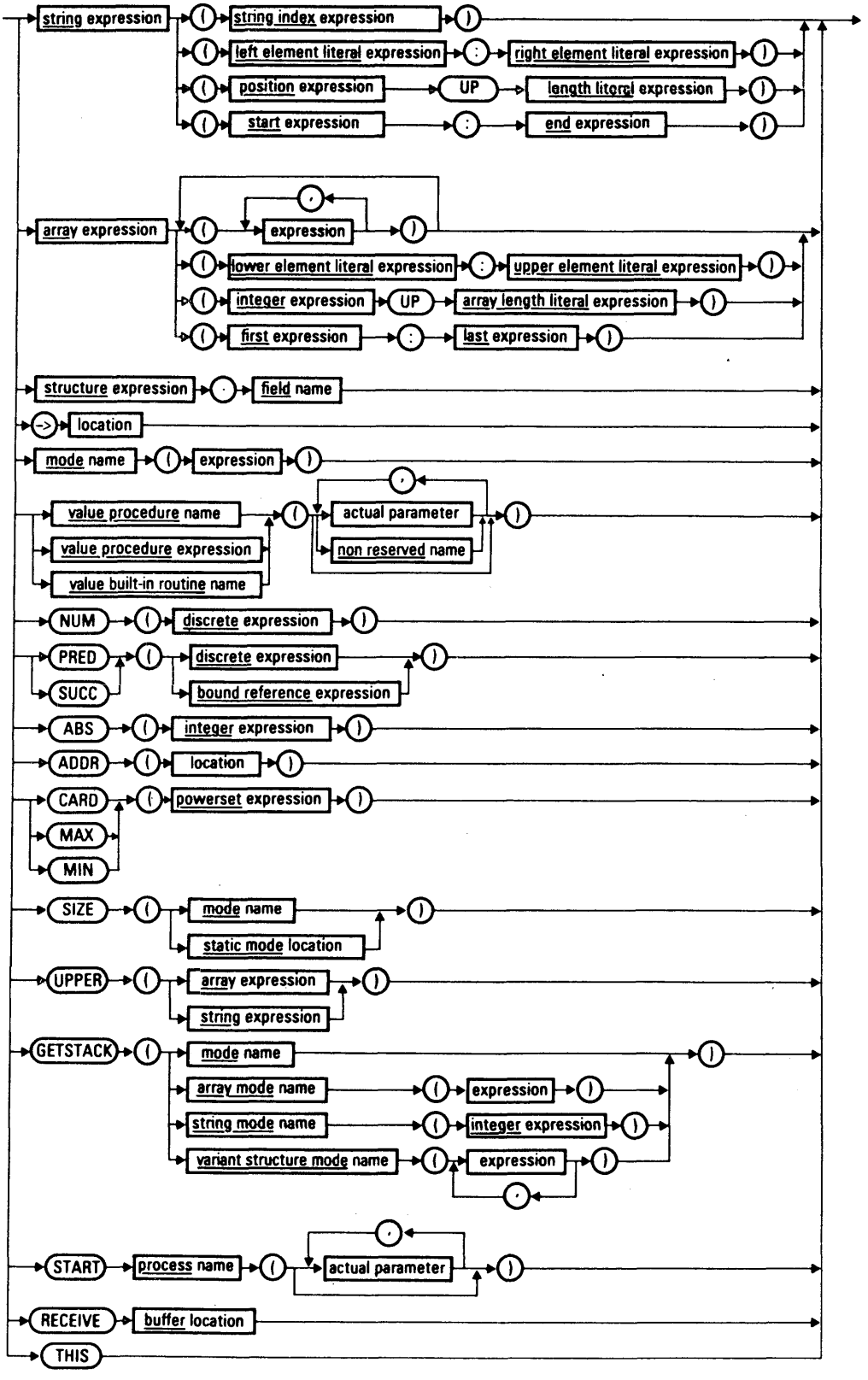
position



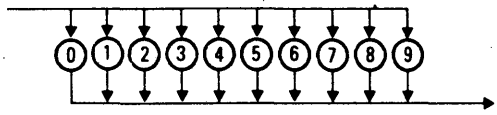

location
access name
reference expression
->
mode name
string location
(
string index expression
)
left element literal expression
:
right element literal expression
position expression
UP
length literal expression
start expression
end expression
array location
index expression
lower element literal expression
upper element literal expression
integer expression
array length literal expression
first expression
last expression
structure location
.
field name
location procedure name
location procedure expression
location built-in routine name
actual parameter
nonreserved name
row expression
mode name
static mode location
value
expression
*
expression
operand 1
OR
XOR
operand 1
operand 1
operand 2
operand 1
AND
operand 2
operand 2
operand 3
operand 2
=
/=
>
>=
<
<=
IN
operand 3
operand 3
operand 4
operand 3
+
-
//
operand 4
operand 4
operand 5
operand 4
*
/
MOD
REM
operand 5
operand 5
operand 6
-
NOT
string repetition expression
operand 6
primitive value
expression


primitive value
location
name
D'
digit
B'
0
1
'
O'
2
3
4
5
6
7
H'
A
B
C
D
E
F
TRUE
FALSE
NULL
C'
letter
symbol
SP
mode name
(:
[
expression
case label list
field name
:)
]
Continued

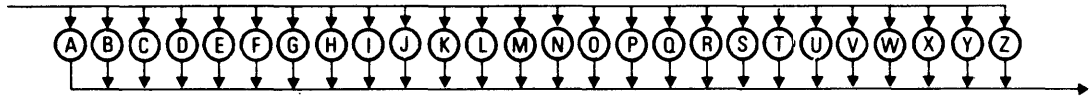
primitive value (continued)



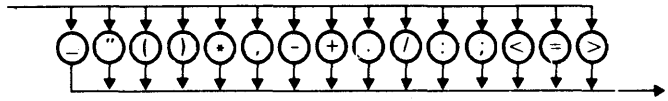
digit



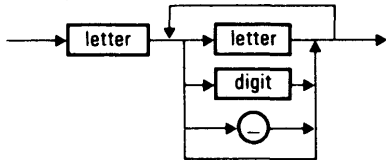
letter



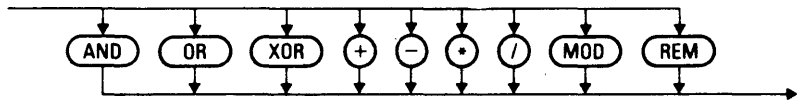
symbol



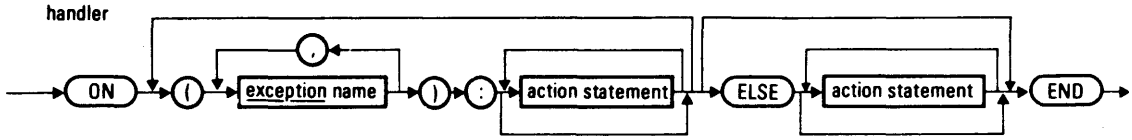
name



dyadic operator



handler



APPENDIX F: INDEX OF PRODUCTION RULES

non-terminal	defined section	page	used on page(s)
<call action>	6.7	113	101
<case action>	6.4	104	101
<case alternative>	6.4	104	104
<case label>	9.1.3	160	159
<case label list>	9.1.3	159	73,159
<case label specification>	9.1.3	159	35,40,104
<case selector list>	6.4	104	104
<cause action>	6.12	117	101
<character>	5.2.4.7	71	13,71
<character mode>	3.4.4	22	20
<character string>	2.4	13	13
<character string literal>	5.2.4.7	71	68
<CHILL directive>	2.6	14	14
<CHILL value built-in routine call>	5.2.16	86	86
<closed dyadic operator>	6.2	102	102
<comment>	2.4	13	
<composite mode>	3.10.1	31	20
<continue action>	6.15	118	101
<control part>	6.5.1	106	106
<data statement>	7.2	129	129
<data statement list>	7.2	129	128,129
<decimal integer literal>	5.2.4.2	69	69
<declaration>	4.1.1	50	50
<declaration statement>	4.1.1	50	129
<defining mode>	3.2.1	17	17
<definition statement>	7.2	129	129
<delay action>	6.16	119	101
<delay alternative>	6.17	119	119
<delay case action>	6.17	119	101
<dereferenced bound reference>	4.2.3	55	53
<dereferenced free reference>	4.2.4	56	53
<dereferenced row>	4.2.15	64	53
<digit>	5.2.4.2	60	12,69,71
<directive>	2.6	14	14
<directive clause>	2.6	14	
<discrete mode>	3.4.1	20	20,25,33,104,107
<do action>	6.5.1	106	101
<dynamic mode location>	4.2.1	53	53

non-terminal	defined section	page	used on page(s)
<element layout>	3.10.6	42	33,41
<element mode>	3.10.3	33	33
<else clause>	6.3	104	103,104
<emptiness literal>	5.2.4.5	70	68
<empty>	6.11	117	117,129
<empty action>	6.11	117	101
<end>	4.2.13	62	62,80
<end bit>	3.10.6	43	42
<end value>	6.5.2	107	107
<entry definition>	7.4	132	132
<entry statement>	7.4	132	129
<event length>	3.9.2	30	30
<event list>	6.17	119	119
<event mode>	3.9.2	30	29
<exception list>	3.7	28	28,131,176
<exception name>	3.7	28	28,117
<exit action>	6.6	112	101
<expression>	5.3.2	92	22,24,30,32,33 35,42,43,55,56 57,58,59,62,63 64,73,79,80,81 83,85,86,89,90 91,92,99,100,103 104,107,111,113 117,119,121,160
<expression conversion>	5.2.14	85	66
<expression list>	4.2.7	58	58,81,86
<field layout>	3.10.6	42	35,40,41
<field name list>	5.2.5	73	73
<fields>	3.10.4	35	35
<first>	4.2.14	63	63,83
<fixed fields>	3.10.4	35	35
<forbid clause>	9.2.6.2	171	171
<forbid name list>	9.2.6.2	171	171
<for control>	6.5.2	107	106
<formal parameter>	7.4	132	131
<formal parameter list>	7.4	131	131,136
<free directive>	2.6	14	14
<free reference mode>	3.6.3	27	26

non-terminal	defined section	page	used on page(s)
<generality>	7.4	132	132
<getstack argument>	5.2.16	86	86
<goto action>	6.9	116	101
<granted element>	9.2.6.2	171	171
<grant statement>	9.2.6.2	171	170
<grant window>	9.2.6.2	171	171
<handler>	10.2	176	50,52,101,131 136,137
<hexadecimal bit string literal>	5.2.4.8	72	72
<hexadecimal digit>	5.2.4.2	69	69,71,72
<hexadecimal integer literal>	5.2.4.2	69	69
<if action>	6.3	103	101
<implementation directive>	---	--	14
<index mode>	3.10.3	33	33,41
<initialisation>	4.1.2	50	50
<instance mode>	3.8	29	20
<integer literal>	5.2.4.2	69	68
<integer mode>	3.4.2	21	20
<irrelevant>	9.1.3	160	159
<iteration>	6.5.2	107	107
<labelled array tuple>	5.2.5	73	73
<labelled structure tuple>	5.2.5	73	73
<last>	4.2.14	63	63,83
<left element>	4.2.6	57	57,79
<length>	3.10.6	43	42
<letter>	5.2.4.7	71	12,71
<level structure mode>	3.10.5	40	35
<lifetime-bound initialisation>	4.1.2	50	50
<literal>	5.2.4.1	68	66
<literal expression list>	3.10.4	35	35
<literal range>	3.4.6	24	24,33,160
<location>	4.2.1	53	56,57,58,59,60 62,63,67,84,86 90,102,107,111 118,119,121,123 124,179
<location built-in routine call>	4.2.11	61	53
<location contents>	5.2.2	67	66
<location conversion>	4.2.12	61	53
<location declaration>	4.1.2	50	50
<location enumeration>	6.5.2	107	107
<location procedure call>	4.2.10	61	53
<loc-identity declaration>	4.1.3	52	50
<loop counter>	6.5.2	107	107
<lower bound>	3.4.6	24	24
<lower element>	4.2.8	59	59,81

non-terminal	defined section	page	used on page(s)
<member mode>	3.5	25	25
<membership operator>	5.3.4	94	94
<mode>	3.3	20	17,28,30,33,35 40,41,50,52,65 146
<mode definition>	3.2.1	17	18,19
<module>	7.6	137	101
<module body>	7.2	129	137
<modulion name>	9.2.6.3	172	172
<monadic operator>	5.3.7	98	98
<multiple assignment action>	6.2	102	102
<name>	2.2	12	14,21,22,23,24 25,26,27,28,29 30,32,33,35,52 54,55,56,60,61 67,70,71,73,83 85,86,89,91, 101,107,111,112 113,116,121,123 124,131,136,137 146,160,171,172 179
<name list>	2.6	14	14,17,35,40,41 50,52,65,123 132
<nested structure mode>	3.10.4	35	35
<newmode definition statement>	3.2.3	19	129
<(n) level alternative>	3.10.5	40	40
<(n) level alternative fields>	3.10.5	40	40
<(n) level fields>	3.10.5	40	40,41
<(n) level fixed fields>	3.10.5	40	40
<(n) level variant fields>	3.10.5	41	40,41
<non-composite mode>	3.3	20	20
<numbered set element>	3.4.5	22	22
<numbered set list>	3.4.5	22	22

non-terminal	defined section	page	used on page(s)
<octal bit string literal>	5.2.4.8	72	72
<octal integer literal>	5.2.4.2	69	69
<on-alternative>	10.2	176	176
<operand-1>	5.3.3	93	92,93
<operand-2>	5.3.4	94	93,94
<operand-3>	5.3.5	95	94,96
<operand-4>	5.3.6	97	95,97
<operand-5>	5.3.7	98	97
<operand-6>	5.3.8	100	98
<operator-3>	5.3.4	94	94
<operator-4>	5.3.5	96	95
<origin array mode name>	3.10.3	33	33
<origin string mode name>	3.10.2	32	32
<origin variant structure mode name>	3.10.4	35	35
<parameter attribute>	3.7	28	28
<parameterised array mode>	3.10.3	33	33
<parameterised string mode>	3.10.2	32	32
<parameterised structure mode>	3.10.4	35	35
<parameter list>	3.7	28	28
<parameter spec>	3.7	28	28,132
<parenthesised expression>	5.3.8	100	100
<pattern size>	3.10.6	42	42
<pos>	3.10.6	42	42
<position>	4.2.6	57	56,57,79
<powerset difference operator>	5.3.5	96	96,102
<powerset enumeration>	6.5.2	107	107
<powerset inclusion operator>	5.3.4	94	94
<powerset mode>	3.5	25	20
<powerset tuple>	5.2.5	73	73
<primitive value>	5.2.1	66	100
<priority>	6.16	119	119,121
<proc body>	7.2	128	131
<procedure attributes>	7.4	132	131
<procedure call>	6.7	113	61,85,113
<procedure definition>	7.4	131	131
<procedure definition statement>	7.4	131	129
<procedure literal>	5.2.4.6	71	68
<procedure mode>	3.7	28	20
<process body>	7.2	129	136
<process definition>	7.5	136	136
<process definition statement>	7.5	136	129
<program>	7.8	138	

non-terminal	defined section	page	used on page(s)
<range>	5.2.5	73	73
<range enumeration>	6.5.2	107	107
<range list>	6.4	104	104
<range mode>	3.4.6	24	20
<reach-bound initialisation>	4.1.2	50	50
<receive buffer case action>	6.19.3	124	122
<receive case action>	6.19.1	122	101
<receive expression>	5.2.18	90	66
<receive signal case action>	6.19.2	123	122
<reference mode>	3.6.1	26	20
<referenced location>	5.2.13	84	66
<referenced mode>	3.6.2	26	26
<region>	7.7	137	129,138
<region body>	7.2	129	137
<relational operator>	5.3.4	94	94
<result>	6.8	115	115
<result action>	6.8	115	101
<result spec>	3.7	28	28,131
<return action>	6.8	115	101
<right element>	4.2.6	57	57,79
<row mode>	3.6.4	27	26
<seized element>	9.2.6.3	172	172
<seize statement>	9.2.6.3	172	170
<seize window>	9.2.6.3	172	172
<send action>	6.18.1	120	101
<send buffer action>	6.18.3	121	120
<send signal action>	6.18.2	121	120
<set element>	3.4.5	23	22
<set list>	3.4.5	22	22
<set literal>	5.2.4.4	70	68
<set mode>	3.4.5	22	20
<signal definition>	8.5	146	146
<signal definition statement>	8.5	146	129
<signal receive alternative>	6.19.2	123	123
<single assignment action>	6.2	102	102
<space>	5.2.4.7	71	71
<start>	4.2.13	62	62,80
<start action>	6.13	118	101
<start bit>	3.10.6	43	42
<start expression>	5.2.17	89	66,118

non-terminal	defined section	page	used on page(s)
<start value>	6.5.2	107	107
<static mode location>	4.2.1	53	52,53,61,86,113 115
<step>	3.10.6	42	42
<step enumeration>	6.5.2	107	107
<step size>	3.10.6	42	42
<step value>	6.5.2	107	107
<stop action>	6.14	118	101
<string concatenation operator>	5.3.5	96	96
<string element>	4.2.5	56	53
<string length>	3.10.2	32	32,57,79
<string mode>	3.10.2	32	27,31
<string repetition operator>	5.3.7	99	98
<string slice>	4.2.13	62	53
<string type>	3.10.2	32	32
<structure field>	4.2.9	60	53
<structure mode>	3.10.4	35	31
<structure tuple>	5.2.5	73	73
<sub-array>	4.2.8	59	53
<sub expression>	5.3.2	92	92
<sub-operand-1>	5.3.3	93	93
<sub-operand-2>	5.3.4	94	94
<sub-operand-3>	5.3.5	95	95
<sub-operand-4>	5.3.6	97	97
<substring>	4.2.6	57	53
<symbol>	5.2.4.7	71	71
<synchronisation mode>	3.9.1	29	20
<synmode definition statement>	3.2.2	18	129
<synonym definition>	5.1	65	65
<synonym definition statement>	5.1	65	129
<tags>	3.10.4	35	35,40
<then clause>	6.3	104	103,104
<tuple>	5.2.5	73	66

non-terminal	defined section	page	used on page(s)
<undefined value>	5.3.1	91	91
<unlabelled array tuple>	5.2.5	73	73
<unlabelled structure tuple>	5.2.5	73	73
<unnamed value>	3.4.5	23	23
<unnumbered set list>	3.4.5	22	22
<upper bound>	3.4.6	24	24
<upper element>	4.2.8	59	59,81
<upper index>	3.10.3	33	33
<value>	5.3.1	91	50,65,73,102 113,115,121,179
<value array element>	5.2.9	81	66
<value array slice>	5.2.11	83	66
<value built-in routine call>	5.2.16	85	66
<value enumeration>	6.5.2	107	107
<value name>	5.2.3	67	66
<value procedure call>	5.2.15	85	66
<value string element>	5.2.6	79	66
<value string slice>	5.2.8	80	66
<value structure field>	5.2.12	83	66
<value sub-array>	5.2.10	81	66
<value substring>	5.2.7	79	66
<variant alternative>	3.10.4	35	35
<variant fields>	3.10.4	35	35
<visibility statement>	9.2.6.1	170	129
<while control>	6.5.3	111	106
<with control>	6.5.4	111	111
<with part>	6.5.4	111	106
<word>	3.10.6	42	42
<zero-adic operator>	5.2.19	91	66

Printed in Switzerland — ISBN 92-61-01121-7