INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

RED BOOK

VOLUME VI – FASCICLE VI.12

CCITT HIGH LEVEL LANGUAGE (CHILL)

RECOMMENDATION Z.200

VIIITH PLENARY ASSEMBLY
MALAGA-TORREMOLINOS, 8-19 OCTOBER 1984

Geneva 1985

INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

RED BOOK

VOLUME VI – FASCICLE VI.12

CCITT HIGH LEVEL LANGUAGE (CHILL)

RECOMMENDATION Z.200

VIIITH PLENARY ASSEMBLY
MALAGA-TORREMOLINOS, 8-19 OCTOBER 1984

Geneva 1985

ISBN 92-61-02251-0

CONTENTS OF THE CCITT BOOK
APPLICABLE AFTER THE EIGHTH PLENARY ASSEMBLY (1984)

RED BOOK

Volume IV – *(4 fascicles, sold separately)*

FASCICLE IV.1 – Maintenance; general principles, international transmission systems, international telephone circuits. Recommendations M.10-M.762 (Study Group IV).

FASCICLE IV.2 – Maintenance; international voice frequency telegraphy and fascimile, international leased circuits. Recommendations M.800-M.1375 (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 0 Recommendations (Study Group IV).

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

Volume VI – *(13 fascicles, sold separately)*

FASCICLE VI.1 – General Recommendations on telephone switching and signalling. Interface with the maritime mobile service and the land mobile services. 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.490 (Study Group XI).

FASCICLE VI.5 – Digital transit exchanges in integrated digital networks and mixed analogue-digital networks. Digital local and combined exchanges. Recommendations Q.501-Q.517 (Study Group XI).

FASCICLE VI.6 – Interworking of signalling systems. Recommendations Q.601-Q.685 (Study Group XI).

FASCICLE VI.7 – Specifications of Signalling System No. 7. Recommendations Q.701-Q.714 (Study Group XI).

FASCICLE VI.8 – Specifications of Signalling System No. 7. Recommendations Q.721-Q.795 (Study Group XI).

FASCICLE VI.9 – Digital access signalling system. Recommendations Q.920-Q.931 (Study Group XI).

FASCICLE VI.10 – Functional Specification and Description Language (SDL). Recommendations Z.101-Z.104 (Study Group XI).

FASCICLE VI.11 – Functional Specification and Description Language (SDL), annexes to Recommendations Z.101-Z.104 (Study Group XI).

FASCICLE VI.12 – CCITT High Level Language (CHILL). Recommendation Z.200 (Study Group XI).

FASCICLE VI.13 – Man-Machine Language (MML). Recommendations Z.301-Z.341 (Study Group XI).

Volume VII – *(3 fascicles, sold separately)*

FASCICLE VII.1 – Telegraph transmission. Series R Recommendations (Study Group IX). Telegraph services terminal equipment. Series S Recommendations (Study Group IX).

FASCICLE VII.2 – Telegraph switching. Series U Recommendations (Study Group IX).

FASCICLE VII.3 – Terminal equipment and protocols for telematic services. Series T Recommendations (Study Group VIII).

Volume VIII – *(7 fascicles, sold separately)*

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

FASCICLE VIII.2 – Data communication networks: services and facilities. Recommendations X.1-X.15 (Study Group VII).

FASCICLE VIII.3 – Data communication networks: interfaces. Recommendations X.20-X.32 (Study Group VII).

FASCICLE VIII.4 – Data communication networks: transmission, signalling and switching, network aspects, maintenance and administrative arrangements. Recommendations X.40-X.181 (Study Group VII).

FASCICLE VIII.5 – Data communication networks: Open Systems Interconnection (OSI), system description techniques. Recommendations X.200-X.250 (Study Group VII).

FASCICLE VIII.6 – Data communication networks: interworking between networks, mobile data transmission systems. Recommendations X.300-X.353 (Study Group VII).

FASCICLE VIII.7 – Data communication networks: message handling systems. Recommendations X.400-X.430 (Study Group VII).

Volume IX – Protection against interference. Series K Recommendations (Study Group V). Construction, installation and protection of cable, and other elements of outside plant. Series L Recommendations (Study Group VI).

Volume X – *(2 fascicles, sold separately)*

FASCICLE X.1 – Terms and definitions.

FASCICLE X.2 – Index of the Red Book.

CONTENTS OF FASCICLE VI.12 OF THE RED BOOK

Part I – Recommendation Z.200

CCITT High Level Language (CHILL)

Part II – Supplements to Recommendation Z.200

PART I

Recommendation Z.200

CCITT High Level Language (CHILL)

Recommendation Z.200

CCITT HIGH LEVEL LANGUAGE (CHILL)

(Geneva, 1984)

CONTENTS

1 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, is contained in the CCITT Manual, 'Formal definition of CHILL'. Another CCITT Manual, 'Introduction to CHILL', serves as an introduction to the language.

This chapter gives an informal description of the facilities of CHILL. The language definition starts at chapter 2.

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 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 of the use 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

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.
input-output modes	association and access modes for input-output operations.

CHILL provides denotations for a set of standard modes. Program defined modes can be introduced by means of **mode definitions**. Some language constructs have a so-called **dynamic mode** attached. A dynamic mode is a mode of which some properties can be determined only dynamically. Dynamic modes are always parameterised modes with run-time parameters. A mode that 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 introduced in the metalanguage only 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, a location has to be **accessed**.

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

There are:

1. location declarations;
2. loc-identity declarations;
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* or *ALLOCATE* built-in routine that 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 have a **read-only** mode, which means that it can only be accessed to obtain a value and not to store a new value into it (except when 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 **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. When the context requires a **literal** or **constant** value, the value is assumed to be evaluated before run-time and therefore cannot generate a run-time exception. A value may be **intra-regional**, in which case it can refer somehow to locations declared within a region. A value may be **composite**, i.e., contain sub-values.

Synonym definition statements establish new names to denote constant values.

1.6 ACTIONS

Actions constitute the algorithmic part of a CHILL program.

The **assignment** action stores a (computed) value into one or more locations. The **procedure call** invokes a procedure, a **built-in routine call** invokes a built-in routine (a built-in routine is a procedure whose definition 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 or a module in a structured manner;

cause action to cause a specific exception;

goto action for unconditional transfer to a labelled program point.

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

To control the concurrent action flow, CHILL provides the **start**, **stop**, **delay**, **continue**, **send**, **delay case** and **receive case** actions and the evaluation of a **receive expression**.

1.7 INPUT AND OUTPUT

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

The input-output reference model knows three states. In the **free state** there is no interaction with the outside world.

Through an *ASSOCIATE* operation the file handling state is entered. In the file handling state there are locations of **association mode**, which denote outside world objects. It is possible via built-in routines to read and modify the language defined attributes of associations, i.e. **existing**, **readable**, **writeable**, **indexable**, **sequencible** and **varying**. File creation and deletion are also done in the file handling state.

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

1.8 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* or *ALLOCATE* built-in routine call), or they can be **implicitly declared** or **generated** as the result of the use of language constructs.

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

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

Modules are provided to restrict the visibility of names to protect against 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 that may be invoked (called) at different places within a program. It may return a value (**value procedure**) or a location (**location procedure**), or deliver no result. In the latter case the procedure can only be called in a procedure call action.

Processes 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 that is considered to be surrounded by an (imaginary) process definition. This outermost process is started by the system under whose control the program is executed.

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

1.9 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 use. The operations **send**, **receive** and **receive case** are defined on buffers; the operations **delay**, **delay case** and **continue** are defined on events. Buffers are a means of synchronising and transmitting information between processes. Events are used only 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 that are shared by several processes.

1.10 GENERAL SEMANTIC PROPERTIES

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

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

The scope rules determine the visibility of names through the program structure and explicit visibility statements. The explicit visibility statements influence the scope of the mentioned names 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.

1.11 EXCEPTION HANDLING

The dynamic semantic conditions of CHILL are those (non context-free) conditions that, in general, cannot be statically determined. (It is left to the implementation to decide whether or not to generate code to test the dynamic conditions at run time.) 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 it is specifiable (i.e., it has a name) and is specified. Whether or not a handler is specified for an exception at a given point can be statically determined. If no explicit handler is specified, control may be transferred to an implementation defined exception handler.

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

1.12 IMPLEMENTATION OPTIONS

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

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

A built-in routine is a procedure whose definition is not specified in CHILL and that has 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.

An implementation defined exception is caused if an implementation defined dynamic condition is violated.

2 PRELIMINARIES

2.1 THE METALANGUAGE

The CHILL description consists of two parts:

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

2.1.1 The context-free syntax description

The context-free syntax is described using an extension of the Backus-Naur Form. Syntactic categories are indicated by one or more English words, written in 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 (|) to denote alternative productions for the non-terminal symbol.

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

Syntactic elements may be grouped together by using curly brackets ({ and }). A curly bracketed group may contain one or more vertical bars, indicating alternative syntactic elements. 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 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 that 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** simple name strings (see Appendix C). These **special** simple name strings are always used with their CHILL meaning and are therefore not influenced by the binding rules of an actual CHILL program.

2.2 VOCABULARY

Programs are represented using the CCITT alphabet no. 5, Recommendation V.3 (see Appendix A1). It is possible to represent any CHILL program using a minimum character set that is a subset of the CCITT alphabet no.5 basic code (see Appendix A2).

The lexical elements of CHILL are:

- special symbols
- simple name strings
- literals.

Apart from the lexical elements there are also special character combinations.The special symbols and special character combinations are listed in Appendix B.

Simple name strings are formed according to the following syntax:

syntax:

<simple name string> ::= *(1)*
<letter> { *<letter>* | *<digit>* | _}* *(1.1)*

The underline character (_) forms part of the simple name string; e.g., the simple name string *life_time* is different from the simple name string *lifetime*. In the case that an alphabet with lower case letters is available, it may be used within simple name strings. Lower case and upper case letters are different, e.g., *Status* and *status* are two different simple name strings.

The language has a number of **special** simple name strings with predetermined meanings, see Appendix 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 and lower case letters is used, the special simple name strings must either all be in upper case representation or all be in lower case representation. The **reserved** simple name strings are only reserved in the chosen representation (e.g., if the lower case fashion is chosen, **row** is reserved, **ROW** is not).
For the **literal qualifications** (see Appendix B) and the letters in a *hexadecimal digit* (see section 5.2.4.2) an implementation must allow either upper case, or both upper and lower case letters.

2.3 THE USE OF SPACES

A space terminates any lexical element or special character combination. Lexical elements are also terminated by the first character that cannot be part of the lexical element. For instance, *IFBTHEN* will be considered a *simple name string* and not as the beginning of an action **IF** *B* **THEN** , //* will be considered as the concatenation symbol (//) followed by an asterisk (*) and not as a divide symbol (/) followed by a comment opening bracket (/*). 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 simple name string list>) *(4.1)*

<simple name string list> ::= *(5)*
<simple name string>{,*<simple name string>*} * *(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 will free the **reserved** simple name strings specified in the *reserved simple name string list* so that they may be redefined.

static properties: A *directive clause* may be inserted at any place 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 10.2).

static conditions: The optional directive-ending symbol (<>) may be omitted only 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 contain neither 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)*

2.7 NAMES AND THEIR DEFINING OCCURRENCES

syntax:

<*name*> ::= *(1)*
<*name string*> *(1.1)*

<*name string*> ::= *(2)*
<*simple name string*> *(2.1)*
| <*prefixed name string*> *(2.2)*

<*prefixed name string*> ::= *(3)*
<*prefix*> ! <*simple name string*> *(3.1)*

<*prefix*> ::= *(4)*
<*simple prefix*> { ! <*simple prefix*> } * *(4.1)*

<*simple prefix*> ::= *(5)*
<*simple name string*> *(5.1)*

<*defining occurrence*> ::= *(6)*
<*simple name string*> *(6.1)*

<*defining occurrence list*> ::= *(7)*
<*defining occurrence*>{, <*defining occurrence*> } * *(7.1)*

<*field name*> ::= *(8)*
<*simple name string*> *(8.1)*

<*field name defining occurrence*> ::= *(9)*
<*simple name string*> *(9.1)*

<*field name defining occurrence list*> ::= *(10)*
<*field name defining occurrence*> {, <*field name defining occurrence*> } * *(10.1)*

<*exception name*> ::= *(11)*
<*simple name string*> *(11.1)*
| <*prefixed name string*> *(11.2)*

<*register name*> ::= *(12)*

<*simple name string*> *(12.1)*
| <*prefixed name string*> *(12.2)*

<*text reference name*> ::= *(13)*
<*simple name string*> *(13.1)*
| <*prefixed name string*> *(13.2)*

<*map reference name*> ::= *(14)*
<*simple name string*> *(14.1)*
| <*prefixed name string*> *(14.2)*

derived syntax: In the reduced character set, /. is used for ! (prefixing operator).

semantics: Names in a program denote objects. Given an occurrence of a *name* (formally: an occurrence of a terminal production of *name*) in a program, the binding rules of section 10.2 provide *defining occurrences* to which that (occurrence of) *name* is bound (formally: occurrences of terminal productions of *defining occurrence*). The *name* then denotes the object defined or declared by the *defining occurrences*. (There can be more than one *defining occurrence* for a *name* only in the case of *set element names* or of *names* with *quasi defining occurrences*.) *Defining occurrences* are said to define the *name*.

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

Exception names are used to identify exception handlers according to the rules stated in Chapter 11.

Register names are used to identify registers in an implementation defined way (see section 8.4).

Text reference names are used to identify pieces of source text in an implementation defined way, subject to the rules in section 8.10.1.

Map reference names are used to specify mapping in an implementation defined way (see section 3.11.6).

When a *name* is bound to more than one *defining occurrence*, each of the *defining occurrences* to which the *name* is bound defines or declares the same object (see 10.2.2 and 8.10 for precise rules).

Each *simple name string* has a canonical name string attached which is the *simple name string* itself.
Each *name string* has a canonical name string attached defined as follows :

- if the *name string* is a *simple name string*, then the canonical name string of that *simple name string*;
- if the *name string* is a *prefixed name string*, then the concatenation in left to right order of all *simple name string*'s in the *name string*, separated by prefixing operators, i.e., interspersed spaces, comments and format effectors (if any) are left out.

In the rest of this document :

- the name string of a *name*, *exception name*, *register name* or *text reference name* is used to denote the canonical name string of the *name string* in that *name*, *exception name*, *register name* or *text reference name* respectively;
- the name string of a *defining occurrence*, *field name* or *field name defining occurrence* is used to denote the canonical name string of the *simple name string* in that *defining occurrence*, *field name* or *field name defining occurrence* respectively.

The binding rules are such that:

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

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

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

- if P is empty, then P ! NS is NS;
- or else, P ! NS is the name string attached to the *prefixed name string* obtained by concatenating all the characters in P, a prefixing operator and all the characters in NS.

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

static properties: A *name* has the static properties attached to the *defining occurrences* to which it is bound. A *field name* has the static properties attached to the *field name defining occurrence* to which it is bound.

3 MODES AND CLASSES

3.1 GENERAL

A location has a mode attached to it; a value has a class attached to it. The mode attached to a location defines the set of values that may be contained in the location, the access methods of the location and the allowed operations on the values. The class attached to a value is a means of determining the modes of the locations that may contain the value. Some values are **strong**. **A strong** value has a class and a mode attached. 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 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 (*&*); e.g., *&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 that are not or cannot be explicitly denoted in the program text but are virtually 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 a mode M, there may exist the **M-value** class. All values with such a class and only those values are **strong** and the mode attached to the value is M.
- For a mode M there may exist 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 or an M-reference class, where M is a dynamic mode.

3.1.3 Properties of, and relations between, modes and classes

Modes in CHILL have properties. These may be hereditary or non-hereditary properties. A hereditary property is inherited from a defining mode to a mode name defined by it. Below a summary is given of the properties that apply to all modes (except for the first, they are all defined in section 10.1) :

1. A mode has a **novelty** (defined in sections 3.2.2, 3.2.3 and 3.3).
2. A mode can have the **read-only property**.
3. A mode can be **parameterisable**.
4. A mode can have the **referencing property**.
5. A mode can have the **tagged parameterised property**.

6. A mode can have the **non-value property**.

Classes in CHILL may have the following properties (defined in section 10.1):

1. A class can have a **root** mode.
2. One or more classes may have a **resulting class**.

Operations in CHILL are determined by the modes and classes of locations and values. This is expressed by the mode checking rules which are defined in section 10.1 as a number of relations between modes and classes. There exists the following relations :

1. Two modes can be **similar**.
2. Two modes can be **v-equivalent**.
3. Two modes can be **equivalent**.
4. Two modes can be **l-equivalent**.
5. Two modes can be **alike**.
6. Two modes can be **N-alike**.
7. Two modes can be **read-compatible**.
8. Two modes can be **dynamic read-compatible**.
9. A mode can be **restrictable** to a mode.
10. A mode can be **compatible** with a class.
11. A class can be **compatible** with a class.

3.2 MODE DEFINITIONS

3.2.1 General

syntax:

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

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

derived syntax: A *mode definition* where the *defining occurrence list* consists of more than one *defining occurrence* is derived from several mode definitions, one for each *defining occurrence*, separated by commas, with the same *defining mode*. For example :

NEWMODE *dollar, pound* = *INT* ;

is derived from

NEWMODE *dollar* = *INT* , *pound* = *INT* ;

semantics: Mode definitions define one or more names to be mode names. Most properties of a mode name are inherited from its defining mode.
Mode definitions occur in synmode and newmode definition statements. A synmode is synonymous with its defining mode. A newmode is not synonymous with its defining mode. The difference is defined in terms of the property novelty, that is used in the mode checking (see section 10.1).

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

Mode names are also the predefined mode names *INT* , *BIN* , *BOOL* , *CHAR* , *PTR* , *INSTANCE* , *ASSOCIATION* and the implementation defined integer mode names (if any, see section 3.4.2).

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

A set of recursive definitions is a set of mode definitions or synonym definitions (see 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 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.

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

examples:

1.15 *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 mode names which are synonymous with their defining mode.

static properties: A *defining occurrence* in a *mode definition* in a *synmode definition statement* defines a **synmode** name (which is also a **mode** name). A **synmode** name is said to be **synonymous** with a 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.

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

examples:

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

3.2.3 Newmode definitions

syntax:

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

semantics: Newmode definition statements define mode names which are not **synonymous** with their defining mode.

static properties: A *defining occurrence* in a *mode definition* in a *newmode definition statement* defines a **newmode** name (which is also a **mode** name).
The **novelty** of the **newmode** name is the *defining occurrence* which defines it.
If the **defining mode** of the **newmode** name is a range mode, then the virtual mode *&name* is introduced as the **parent** mode of the **newmode** name. The **defining mode** of *&name* is the **parent** mode of the range mode, and the **novelty** of *&name* is that of the **newmode** name.

examples:

11.6 **NEWMODE** *line = INT (1:8);* *(1.1)*
11.12 **NEWMODE** *board* = **ARRAY** *(line)* **ARRAY** *(column) square;* *(1.1)*

3.3 MODE CLASSIFICATION

syntax:

<*mode*> ::= *(1)*
[**READ**] <*non-composite mode*> *(1.1)*
| [**READ**] <*composite mode*> *(1.2)*

<*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)*
| <*input-output mode*> *(2.7)*

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

static properties: A mode has the following hereditary properties :

- It is a **read-only** mode if it is an explicit or an implicit **read-only** mode.
- It is an explicit **read-only** mode if **READ** is specified or it is a **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 is a **read-only** mode.
- It is an implicit **read-only** mode if
  - it is the **element** mode of a **read-only** array mode (see section 3.11.3);
  - it is a **field** mode of a **read-only** structure mode or it is the mode of a **tag field** of a **parameterised** structure mode (see section 3.11.4).

Read-only modes have the same properties as their corresponding non-**read-only** modes except for the **read-only property** (see section 10.1.1).

A mode has the following non-hereditary properties :

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

  The **novelty** of a mode that is a *mode name* (**READ** *mode name*) is defined in sections 3.2.2 and 3.2.3.
- It has a **size** that is the value delivered by *SIZE (M)*, where *M* is a virtual **synmode** name **synonymous** with the *mode*.

3.4 DISCRETE MODES

3.4.1 General

syntax:

<discrete mode> ::= (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 that are not range modes can be parent modes of range mode (see section 3.4.6). All discrete modes define an upper bound and a lower bound and a number of values.

3.4.2 Integer modes

syntax:

<integer mode> ::= (1)
  INT (1.1)
| *BIN* (1.2)
| *<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). An implementation may define other integer modes with different bounds (e.g., *LONG_INT*, *SHORT_INT*, ...) that may also be used as parent modes for ranges (see section 12.2).

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

- The **upper bound** and **lower bound** are the literals denoting respectively the highest and lowest value defined by the integer mode. They are implementation defined.
- The **number of values** which is: **upper bound** − **lower bound** + 1.

examples:

1.5 *INT* *(1.1)*

3.4.3 Boolean modes

syntax:

<boolean mode> ::= *(1)*
BOOL *(1.1)*
| *<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). *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)*
CHAR *(1.1)*
| *<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** 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)*
SET (*<set list>*) *(1.1)*
| *<set mode name>* *(1.2)*

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

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

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

<unnumbered set list> ::= *(5)*

<set element> { ,<set element>} * (5.1)

<set element> ::= (6)
<defining occurrence> (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 defined by *defining occurrences* in the *set list*; the unnamed values are the other values. The internal representation of the named values is the integer value associated with the named value (see below). This representation also defines the ordering of the values.

static properties: A *defining occurrence* in a *set list* defines a **set element** name.

A set mode has the following hereditary properties:

- A set mode has a set of **set element** names which is the set of names defined by *defining occurrences* in its *set list*.
- Each **set element** name of a set mode has an 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 *defining occurrence* defining 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* has representation value 3 attached.
- A set mode has an **upper bound** and a **lower bound** which are its **set element** names with the highest and lowest representation 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 its **set element** names 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:

11.7 **SET** *(occupied, free)* (1.1)
6.3 *month* (1.2)

3.4.6 Range modes

syntax:

<range mode> ::= (1)
<discrete mode name>(<literal range>) (1.1)
| **RANGE** (<literal range>) (1.2)
| *BIN* (<integer literal expression>) (1.3)
| <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 that determines the operations on and ordering of the range values.

static properties: A range mode has the following non-hereditary property: it has a **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** that 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 that is less than or equal to the value delivered by *upper bound*, and both values must belong to the set of values defined by *discrete mode name*, if specified.

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

examples:

9.5	*INT (2:max)*	*(1.1)*
11.12	*line*	*(1.4)*
9.5	*2:max*	*(2.1)*

3.5 POWERSET MODES

syntax:

<*powerset mode*> ::= (1)
POWERSET <*member mode*> (1.1)
| <*powerset mode name*> (1.2)

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

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

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

- It has a **member** mode that is the *member mode.*

examples:

8.4	**POWERSET** *CHAR*	*(1.1)*
9.5	**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 4.2.5), delivering the location that is referenced.

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

3.6.2 Bound reference modes

syntax:

<bound reference mode> ::=	*(1)*
REF *<referenced mode>*	*(1.1)*
| *<bound reference mode name>*	*(1.2)*
<referenced mode> ::=	*(2)*
<mode>	*(2.1)*

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

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

- It has a **referenced** mode that is the *referenced mode.*

examples:

10.42	**REF** *cell*	*(1.1)*

3.6.3 Free reference modes

syntax:

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

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

examples:

19.8 *PTR* *(1.1)*

3.6.4 Row modes

syntax:

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

static condition: The *variant structure mode name* must be **parameterisable**.

examples:

8.6 **ROW** *CHAR (max)* *(1.1)*

3.7 PROCEDURE MODES

syntax:

<procedure mode> ::= *(1)*
PROC ([*<parameter list>*]) [*<result spec>*]
[**EXCEPTIONS** (*<exception list>*)] [**RECURSIVE**] *(1.1)*
| *<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** [**DYNAMIC**] *(4.1)*

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

<result attribute>::= *(6)*
[**NONREF**] **LOC** [**DYNAMIC**] *(6.1)*

<exception list> ::= *(7)*
<exception name> { ,*<exception name>*} * *(7.1)*

derived syntax: A *result spec* without the optional **reserved** simple name string **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 that 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 result 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 those 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 **non-value property**.

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

3.8 INSTANCE MODES

syntax:

<instance mode> ::= *(1)*
INSTANCE *(1.1)*
| *<instance mode name>* *(1.2)*

semantics: An instance mode defines values which uniquely identify processes. The creation of a new process (see sections 5.2.14, 6.13 and 9.1) yields a unique instance value as identification for the created process.

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

examples:

15.39 *INSTANCE* *(1.1)*

3.9 SYNCHRONISATION MODES

3.9.1 General

syntax:

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

semantics: Locations of synchronisation mode provide a means of synchronisation and communication between processes (see chapter 9). 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)*
EVENT [*(<event length>)*] *(1.1)*
| *<event mode name>* *(1.2)*

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

semantics: Event mode locations provide a 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 an optional **event length**, which is the value delivered by *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)*
BUFFER [*(<buffer length>)*]*<buffer element mode>* *(1.1)*
| *<buffer mode name>* *(1.2)*

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

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

N.B. The syntax given above is syntactically ambiguous in connection with the syntax of the array modes. The following default interpretation applies: if **BUFFER** is immediately followed by an opening parenthesis, the text immediately following 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 a 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 5.3.8 respectively.

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

- It has an optional **buffer length**, that is the value delivered by *buffer length*.
- It has a **buffer element** mode, that is the *buffer element mode*.

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

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

examples:

16.30 **BUFFER** *(1) user_messages* *(1.1)*
16.34 *user_buffers* *(1.2)*

3.10 INPUT-OUTPUT MODES

3.10.1 General

syntax:

<input-output mode> ::= *(1)*
<association mode> *(1.1)*
| *<access mode>* *(1.2)*

semantics: Input-output modes are used for input-output operations as defined in chapter 7. There exist no expression in CHILL denoting a value defined by an input-output mode. As a consequence there are no operations defined on the values.

examples:

20.17 ASSOCIATION *(1.1)*

3.10.2 Association modes

syntax:

<association mode> ::= *(1)*
ASSOCIATION *(1.1)*
| *<association mode name>* *(1.2)*

semantics: Association mode locations can contain a value which represents a relation to an outside world object. Such a relation is called an association in CHILL; associations can be created by the built-in routine *ASSOCIATE* and be ended by *DISSOCIATE* .

3.10.3 Access modes

syntax:

<access mode> ::= *(1)*
ACCESS [*(<index mode>)*] [*<record mode>* [**DYNAMIC**]] *(1.1)*
| *<access mode name>* *(1.2)*

<record mode> ::= *(2)*
<mode> *(2.1)*

<index mode> ::= *(3)*
<discrete mode> *(3.1)*
| *<literal range>* *(3.2)*

N.B. The syntax given above is syntactically ambiguous in connection with the syntax of the array mode. The following default interpretation applies : if **ACCESS** is immediately followed by an opening parenthesis, the text following is considered to be the start of the optional *index mode* denotation and not as belonging to the *record mode*.

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

semantics: Access mode locations provide the means for positioning a file and for tranferring values from the CHILL program to a file in the outside world, and vice versa.

An access mode may define a *record mode*; this record mode defines the **root** mode of the class of the values that can be transferred via a location of that access mode to or from a file. The mode of the transferred value may be dynamic, i.e., the size of the record may vary, when the attribute **DYNAMIC** is specified in the access mode denotation.

An access mode may also define an *index mode*; such an index mode defines the size of a "window" to (a part of) the file, from which it is possible to read (or write) records randomly. Such a window

can be positioned in an (indexable) file by the connect operation. If no *index mode* is specified, then it is possible to transfer records only sequentially.

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

- It has an optional **record** mode which is the *record mode* if present. It is a **dynamic record** mode if **DYNAMIC** is specified, or a **static record** mode, otherwise.
- It has an optional **index** mode, which is the *index mode*.

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

If **DYNAMIC** is specified, the **record** mode must be **parameterisable**.

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

examples:

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

3.11 COMPOSITE MODES

3.11.1 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 4.2.6-10 and 5.2.6-10).

3.11.2 String modes

syntax:

<string mode> ::= *(1)*
<string type>(<string length>) *(1.1)*
| *<parameterised string mode>* *(1.2)*
| *<string mode name>* *(1.3)*

<parameterised string mode> ::= *(2)*
<origin string mode name>(<string length>) *(2.1)*
| *<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.

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

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 *string length*.
- It has an **upper bound** and a **lower bound** which are the values delivered by *string length* – *1* and *0* respectively.

A string mode is **parameterised** if and only if it is a *parameterised string mode*.

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

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:

7.51 *CHAR (20)* *(1.1)*

3.11.3 Array modes

syntax:

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

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

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

<*upper index*> ::= *(4)*
<*discrete literal* expression> *(4.1)*

<*element mode*> ::= *(5)*
<*mode*> *(5.1)*

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

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

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 3.11.6). 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** that 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. The **element** mode is an implicit **read-only** mode if it is **READ** *M*.

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

An array mode is **parameterised** if and only if it is a *parameterised array mode*.

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

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

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

examples:

5.29	**ARRAY** *(1:16)* **STRUCT** *(c4, c2, c1 BOOL)*	*(1.1)*
11.12	**ARRAY** *(line)* **ARRAY** *(column) square*	*(1.1)*
11.17	*board*	*(1.3)*

3.11.4 Structure modes

syntax:

<structure mode> ::= *(1)*
<nested structure mode> *(1.1)*
| *<level structure mode>* *(1.2)*
| *<parameterised structure mode>* *(1.3)*

| <*structure mode* name> (1.4)

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

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

<*fixed fields*> ::= (4)
<*field name defining occurrence 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)
<*field name defining occurrence list*> <*mode*> [<*field layout*>] (8.1)

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

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

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

derived syntax: A *level structure mode* is derived syntax for a *nested structure mode*. This is explained in section 3.11.5.

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

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 that is attached to the component name. Structure values may reside in (composite) structure locations, where the component name serves as an access to the sub-location. The components of a structure value or location are called **fields** and their names **field** names.

There are **fixed structures**, **variant structures** and **parameterised structures**.

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

Variant structures have variant fields, i.e., fields that are not always present. For tagged variant structures, the presence of these fields is known only at run time from the value(s) of certain

associated fixed field(s) called **tag** fields. Tag-less variant structures do not have **tag** fields. Because the composition of a variant structure may change during run time, the size of a variant structure location is based upon the largest choice (worst case) of variant alternatives.

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 (virtual) dynamic **parameterised** structure modes also exist. Their semantics are defined in section 3.12.4.

The layout of a structure location or value can be controlled by means of a field layout specification (see section 3.11.6).

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

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* that 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 defined below for the different cases. A name is said to be a **field** name if and only if it is defined in a *field name defining occurrence list* in *fixed fields* or *variant fields* in a *structure mode.*

  Each **field** name of a structure mode has a **field** mode attached that is either *M* or **READ** *M*, where *M* is the mode following the **field** name. The **field** mode is **READ** *M* if *M* is not a **read-only** mode and either the structure mode is a **read-only** mode, or the field is a **tag-field** of a **parameterised** structure mode. The **field** mode is an implicit **read-only** mode if it is **READ** *M*.

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

- A *structure mode* denotes a **mapped** mode if and only if its **field** names have a field layout that 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 *field name defining occurrence list* in *fixed fields.* These **field** names are **fixed field** names.

variant structures:

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

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

occurrence list in *alternative fields*. **Field** names defined by a *field name defining occurrence list* in *fixed fields* are the **fixed field** names of the **variant** structure mode; its other **field** names are the **variant field** names.

A **field** name of a **variant** structure mode is a **tag field** name if and only if it occurs in any *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 *field name defining occurrence 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 follows:
  - if it is a **tagged variant** structure mode, the list of M_i -value classes, where M_i are the modes of the **tag field** names in the order that they are defined in *fixed fields*;
  - if it is a **tag-less variant** structure mode, the list is built up from the individual **resulting lists of classes** of each *alternative 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 10.1.3).

parameterised structures:

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

A **parameterised** structure mode has an **origin variant** structure mode which is the mode denoted by *origin variant structure mode name*.

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

- It has an **origin variant** structure mode that 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 that is the union of the set of **fixed field** names of its **origin variant** structure mode and the set of those **variant field** names of its **origin variant** structure mode that are defined in *variant alternative* occurrences that are selected by the list of values defined by *literal expression list*.

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

The *mode* of *variant fields* may have neither the **non-value property** nor the **tagged parameterised property**.

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 each *variant alternative* occurrence.

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

For *alternative fields*, the case selection conditions must be fulfilled (see section 10.1.3), 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)
11.7    STRUCT (status SET (occupied, free),
                    CASE status OF
                      (occupied): p piece,
                      (free):
                    ESAC )                                          (2.1)
2.6     fraction                                                    (1.4)
11.7    status SET (occupied, free)                                 (4.1)
11.8    status                                                      (7.1)
11.9    p piece                                                     (8.1)
```

3.11.5 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 <field name defining occurrence list> <mode> [ <field layout> ]   (3.1)
      | n <field name defining occurrence 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 <field name defining occurrence list> <mode> [*<field layout>*] (6.1)
| *n <field name defining occurrence 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 3.11.4).

If a structure contains fields that 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, parameter and result specifications, the 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.

READ may not be specified immediately in front of a *level structure mode*.

examples:

```
19.12   DCL 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)
```

3.11.6 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> ])                    (3.1)

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

<word> ::=                                              (5)
        <integer literal expression>                    (5.1)
      | <map reference name>                            (5.2)

<step size> ::=                                         (6)
        <integer literal expression>                    (6.1)

<start bit> ::=                                         (7)
        <integer literal expression>                    (7.1)

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

<length> ::=                                            (9)
        <integer literal expression>                    (9.1)
```

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 is either **PACK** or **NOPACK** , mapping information is either *step* in the case of array modes, or *pos* in the case 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 fields 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 **referable**.

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

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

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

defines a bit-offset of

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

and a length of *NUM (length)* bits, where *WIDTH* is the (implementation defined) number of bits in a word, and *word* is either an *integer literal expression* or a *map reference name* delivering an implementation defined integer value.

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

A *step* of the form

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

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

$b_i = i * NUM\ (step\ size)$.

The jth element of the array starts at a bit-offset of $p + b_j - 1$ from the start of each location of the array mode, where p is the bit-offset specified in *pos*. Each element occupies the length given in *pos*.

Defaults

The notation:

POS *(<word 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>)*

is semantically equivalent to

STEP *(<pos> , SSIZE)*

where *SSIZE* is the *<length>* specified in *pos* or derivable from *pos* by the above rules.

static properties: For any location of an array mode the element layout of the mode determines the referability of its sub-locations (including sub-arrays, array slices) as follows:

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

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

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

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

Each of *word*, *start bit*, *end bit*, *length* and *step size* must, if specified, deliver a non negative value; and the values delivered by *start bit* and *end bit* must be less than *WIDTH*, the number of bits in an implementation's word; and the value delivered by *start bit* must be less than or equal to that of *end bit*.

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

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

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

Consistency and feasibility

Consistency:

No component of a structure may be specified such that it occupies any bits occupied by another component of the same object except in the case of two **variant field** names defined in the same *alternative 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 that themselves have **referable** components.

examples:

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

3.12 DYNAMIC MODES

3.12.1 General

A dynamic mode is a mode of which some properties 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.

3.12.2 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 **string length** which is the value delivered by *integer expression*.

3.12.3 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 (discrete expression) – NUM (lower bound) + 1

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

3.12.4 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 that 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 that is the list of values delivered by the expressions in the *expression list*.

4 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)*
11.36 *starting_square* **LOC** *:= b(m.lin_1)(m.col_1)* *(2.2)*

4.1.2 Location declarations

syntax:

<location declaration> ::= *(1)*
<defining occurrence 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 defining occurrences specified in the *defining occurrence list.*

With *reach-bound initialisation*, the *value* is evaluated each time the reach in which the declaration is placed is entered (see section 8.2) and the delivered value is assigned to the location(s). Before the *value* is evaluated the location(s) contain(s) an **undefined** value.

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 8.2 and 8.9).

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

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

- tagged parameterised property : the created **tag** field sub-location(s) are initialised with their corresponding parameter value.
- non-value property :

- the created event and/or buffer (sub)-location(s) are initialised to "empty", i.e., no delayed processes are attached to the event or buffer nor are there messages in the buffer.

- the created association (sub)-location(s) are initialised to "empty", i.e., they do not contain an association.

- the created access (sub)-location(s) are initialised to "empty", i.e., they are not connected to an association.

The semantics of **STATIC** and *handler* can be found in section 8.9 and chapter 11, respectively.

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

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

If the *mode* has the **read-only property**, *initialisation* must be specified. If the *mode* has the **non-value 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:

5.7 *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)*
<defining occurrence list> *<mode>* **LOC** [**DYNAMIC**] *<assignment symbol>*
<location> [*<handler>*] *(1.1)*

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

If the *location* is evaluated dynamically, this evaluation is done each time the reach in which the loc-identity declaration is placed is entered. In this case, a declared name denotes an **undefined** location prior to the first evaluation during the lifetime of the access denoted by the declared name (see sections 8.2 and 8.9).

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

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

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

examples:

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

4.1.4 Based declarations

syntax:

<based declaration> ::= *(1)*
<defining occurrence 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* ;

semantics: A based declaration with *bound or free reference location name* specifies as many access names as there are defining occurrences in the *defining occurrence 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: A defining occurrence in a *based declaration* with *bound or free reference location name* defines a **based** name. The mode attached to a **based** name is the *mode* specified in the *based declaration*. A **based** name is **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:

19.12 *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)*
<access name> *(1.1)*
| *<dereferenced bound reference>* *(1.2)*
| *<dereferenced free reference>* *(1.3)*
| *<dereferenced row>* *(1.4)*
| *<string element>* *(1.5)*
| *<string slice>* *(1.6)*
| *<array element>* *(1.7)*
| *<array slice>* *(1.8)*
| *<structure field>* *(1.9)*
| *<location procedure call>* *(1.10)*
| *<location built-in routine call>* *(1.11)*
| *<location conversion>* *(1.12)*

semantics: A location is an object that can contain values. Locations have to be accessed to store or obtain a value.

static properties: A *location* has the following properties :

- It can be **static** or not (see section 8.9)
- It can be **intra-regional** or **extra-regional** (see section 9.2.2);
- It can be **referable** or not. The language definition requires certain locations to be **referable** as defined in the appropriate sections. An implementation may extend referability to other locations (see chapter 12);
- It has a **mode**, as defined in the appropriate sections. This mode is either static or dynamic.

dynamic conditions: 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 a *location enumeration*;
- a **location do-with** name : i.e., a **field** name used as direct access in the *do action* with a *with part*.

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

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

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

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

When accessing via a *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:

4.12 a (1.1)
11.39 starting (1.2)
19.17 x (1.3)
15.35 each (1.4)
5.10 c1 (1.5)

4.2.3 Dereferenced bound references

syntax:

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

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

static properties: The mode attached to a *dereferenced bound reference* is the *mode name* if one is specified, otherwise the **referenced** mode of the mode of the *bound reference primitive value*. A *dereferenced bound reference* is **referable**.

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

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

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

examples:

10.54 p -> (1.1)

4.2.4 Dereferenced free references

syntax:

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

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

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

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

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

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

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

4.2.5 Dereferenced rows

syntax:

<*dereferenced row*> ::= (1)
<*row primitive value*> -> (1.1)

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

static properties: The dynamic mode attached to a *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 primitive value* and where the parameters are, depending on the **referenced origin** mode:

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

A *dereferenced row* is **referable**.

static conditions: The *row primitive value* must be **strong**.

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

The *EMPTY* exception occurs if the *row primitive value* delivers *NULL* .

examples:
8.10 input -> *(1.1)*

4.2.6 String elements

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

derived syntax: A *string element* is derived syntax for a *string slice* of length 1 (see section 4.2.7); e.g.,
<string location> (<start element>)
is derived from:
<string location> (<start element> **UP** *1)*

examples:
18.16 string ->(i) *(1.1)*

4.2.7 String slices

syntax:
<string slice> ::= *(1)*
<string location> (<left element> : <right element>) *(1.1)*
| *<string location> (<start element>* **UP** *<slice size>)* *(1.2)*

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

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

<start element> ::= *(4)*
<integer expression> *(4.1)*

<slice size> ::= *(5)*
<integer expression> *(5.1)*

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

static properties: The (possibly dynamic) mode attached to a *string slice* is a **parameterised** string mode constructed as:
&name (string size)
where *&name* is a virtual **synmode** name **synonymous** with the (possibly dynamic) mode of the *string location* and where *string size* is either

NUM (right element) – NUM (left element) + 1

or

NUM (slice size).

The mode attached to a *string slice* is static if *string size* is **literal** : i.e., *left element* and *right element* are **literal** or *slice size* is **literal**; otherwise, the mode is dynamic.

static conditions: If *left element* and *right element* are **literal** or *slice size* is **literal**, then they must deliver integer values such that the following relations hold:

$0 \leq$ *NUM (left element)* $\leq$ *NUM (right element)* $\leq L - 1$

$1 \leq$ *NUM (slice size)* $\leq L$

where L is the **string length** of the *string location*. If the mode of *string location* is dynamic, these relations can only be checked at run time; see below.

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

$0 \leq$ *NUM (left element)* $\leq$ *NUM (right element)* $\leq L - 1$

$0 \leq$ *NUM (start element)* $<$ *NUM (start element)* $+$ *NUM (slice size)* $\leq L$

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

examples:

18.26 blanks –> (count : 9) *(1.1)*
18.23 string –>(scanstart **UP** *10)* *(1.2)*

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

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

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

$$L \leq \textit{expression} \leq U$$

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

examples:

11.36 b(m.lin_1)(m.col_1) *(1.1)*

4.2.9 Array slices

syntax

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

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

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

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

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

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

&name(upper index)

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

NUM (upper index) = NUM (L) + NUM (upper element) - NUM (lower element)

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

NUM (upper index) = NUM (L) + NUM (slice size) - 1

where L is the **lower bound** of the mode of the array *location*.

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

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

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

If *lower element* and *upper element* are both **literal**, or if *slice size* is **literal**, they must deliver values such that the following relations hold:

$$L \leq \textit{lower element} \leq \textit{upper element} \leq U$$

$1 \leq NUM\ (\ slice\ size\ ) \leq NUM\ (U) - NUM\ (L) + 1$

where L and U are respectively the **lower bound** and **upper bound** of the mode of the *array location*. If the mode of *array location* is dynamic, these relations can only be checked at run time; see below.

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

$L \leq lower\ element \leq upper\ element \leq U$

$NUM\ (L) \leq NUM\ (\ first\ element\ ) \leq NUM\ (\ first\ element\ ) + NUM\ (\ slice\ size\ ) - 1 \leq NUM\ (U)$

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

examples:

17.27 res (0 : count – 1) *(1.1)*

4.2.10 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 **referable** if the **field layout** of the *field name* is **NOPACK** .

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

dynamic conditions: 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:

10.57 last ->.info *(1.1)*

4.2.11 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* if **DYNAMIC** is not specified in it; otherwise it is a dynamically parameterised version of it that has the same parameters as the mode of the delivered location.

The *location procedure call* is **referable** if **NONREF** is not specified in the **result spec** of the *location procedure call*.

dynamic conditions: The *location procedure call* must not deliver an **undefined** location and the lifetime of the delivered location must not have ended.

4.2.12 Location built-in routine calls

syntax:

<location built-in routine call> ::= (1)
<implementation location built-in routine call> (1.1)
| *<CHILL location built-in routine call>* (1.2)

<CHILL location built-in routine call> ::= (2)
<io CHILL location built-in routine call> (2.1)

semantics: A location is delivered as the result of an implementation location built-in routine call or a CHILL location built-in routine call. For the io CHILL location built-in routine call see section 7.4.

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

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

4.2.13 Location conversions

syntax:

<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.*

A *location conversion* is **referable.**

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

The following relation must hold:

SIZE (mode name) = SIZE (static mode location)

5 VALUES AND THEIR OPERATIONS

5.1 SYNONYM DEFINITIONS

syntax:

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

<synonym definition> ::= *(2)*
<defining occurrence list> [*<mode>*] = *<constant value>* *(2.1)*

derived syntax: A *synonym definition*, where *defining occurrence list* consists of more than one defining occurrence, is derived from several *synonym definition* occurrences, one for each defining occurrence 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 that denotes the specified constant value.

static properties: A *defining occurrence* in a *synonym definition* defines 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:

1.17 **SYN** *neutral_for_add = 0,*
neutral_for_mult = 1; *(1.1)*
2.18 *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 string slice>* *(1.6)*
| *<value array element>* *(1.7)*
| *<value array slice>* *(1.8)*
| *<value structure field>* *(1.9)*
| *<expression conversion>* *(1.10)*
| *<value procedure call>* *(1.11)*
| *<value built-in routine call>* *(1.12)*

| <*start expression*> (1.13)
| <*zero-adic operator*> (1.14)
| <*parenthesised expression*> (1.15)

semantics: A primitive value is the basic constituent of an expression. Some primitive values have a dynamic class, i.e. a class based on a dynamic mode. For these primitive values the compatibility checks can only be completed at run time. Check failure will then result in the *TAGFAIL* or *RANGEFAIL* exception.

static properties: The class of the *primitive value* is the class of the *location contents*, *value name*, ...etc., respectively.

A *primitive value* is **constant** if and only if it is a *value name*, *literal*, *tuple*, *referenced location*, *expression conversion* or *value built-in routine call* that is **constant**

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

5.2.2 Location contents

syntax:

<*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 **non-value property**.

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

examples:

3.7 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)
| <*general procedure name*> (1.5)

semantics: A value name delivers a value.

A value name is one of the following:

- a **synonym** name : i.e., a name defined in a *synonym definition statement*;
- a **value enumeration** name : i.e., a name defined by a *loop counter* in a *value enumeration*;
- a **value do-with** name : i.e., a **field** name introduced as value name in the *do action* with a *with part*;
- a **value receive** name : i.e., a name introduced in a *receive case action*;
- a **general procedure** name (see section 8.4).

static properties: The class of a *value name* is the class of the *synonym name*, *value enumeration name*, *value do-with name*, *value receive name* or the M-derived class, where M is the mode of the *general procedure name*, respectively.

A *value name* is **literal** if and only if it is a *synonym name* that is **literal**.

A *value name* is **constant** if it is a *synonym name* or a *general procedure name* denoting a **procedure** name which has attached a *procedure definition* which is not surrounded by a block.

static conditions: The *synonym name* must not be **undefined**.

dynamic conditions: Evaluating a *value do-with name* causes a *TAGFAIL* exception if the denoted value is a **variant** field 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:

10.12	*max*	*(1.1)*
8.8	*i*	*(1.2)*
15.54	*this_counter*	*(1.4)*

5.2.4 Literals

5.2.4.1 General

syntax:

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

semantics: A literal delivers a constant value.

static properties: The class of the *literal* is the class of the *integer literal*, *boolean literal*, ...etc, respectively. A *literal* is **discrete** if it is either an *integer literal*, a *boolean literal*, a *set literal*, a *character string literal* of length 1 or a *bit string literal* of length 1.

The letter together with the following apostrophe which starts an *integer literal*, *boolean literal*, *character string literal* and *bit string literal* (i.e. *B'*, *C'*, *D'*, *H'*, *O'*) is a **literal qualification**.

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' { <octal digit> | _ } $^{+}$ *(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)*

<octal digit> ::= *(8)*
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 *(8.1)*

N.B. : An implementation may support lower case letters for the **literal qualifications** (i.e. *d'*, *b'*, *o'*, *h'*) and for the letters in *hexadecimal digit* (i.e. *a*, *b*, *c*, *d*, *e*, *f*).

semantics: An integer literal delivers a non-negative integer value. The usual decimal (base 10) notation is provided as well as binary (base 2), octal (base 8) and hexadecimal (base 16). The underline character (_) is not significant : i.e., it serves only for readability and it does not influence the denoted value.

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 characters.

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)*
11.78	*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:

10.43 NULL (1.1)

5.2.4.6 Character string literals

syntax:

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

N.B. : *SP* denotes the character "space"; see Appendix A1. An implementation may support lower case letters for the **literal qualifications** (i.e. *c'*).

semantics: A character string literal delivers a character string value that 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 character that is in the CCITT alphabet no. 5, as terminal production of *<character>* (see Appendix A1). Apart from the printable representation, the hexadecimal representation may be used. In this case, each *octal digit* or *hexadecimal digit* pair denotes that character value whose representation corresponds to the given hexadecimal value (see Appendix A1); the underline character (_) is not significant.

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

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

examples:

8.19 *'A-B<ZAA9K' ' '* (1.1)
8.19 ' ' (6.1)

5.2.4.7 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' { <octal digit> | _ } *'* (3.1)

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

N.B. : An implementation may support lower case letters for the **literal qualifications** (i.e. *b'*, *o'*, *h'*).

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

static properties: The **length** of a *bit string literal* is either the number of *0* and *1* occurrences after *B'*, three times the number of *octal digit* 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)
| *<character string literal>* (1.2)
| *<bit string literal>* (1.3)

<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)

N.B. : If *tuple* is a *character string literal* or a *bit string literal*, the syntactic construct is ambiguous; it will be interpreted as a *tuple* if and only if a *character string literal* or *bit string literal* occurs in a context where an *array tuple* without *mode name* is legal.

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 10.1.3).

If an array tuple is **constant** and the element mode is **compatible** with the *CHAR(1)*-derived (*BIT(1)*-derived) class, it is allowed to use a *character* (*bit*) *string literal* as a shorthand for the array tuple (e.g. *(:'a', 'b', 'c', 'd' :)* may be written as *'abcd'*).

If it is a structure value, it is a (possibly labelled) set of values for the fields of the structure. In the unlabelled structure tuple, the values are given for the fields in the same order as they are specified in the attached structure mode. In the labelled structure tuple, the values are given for the fields whose field names are specified in the field name list for the value.

The order of evaluation of the expressions and values in a tuple is undefined and they may be considered as being evaluated in 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 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* or in a *start expression*, 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;
- If the *tuple* is the *value* in a *GETSTACK* or *ALLOCATE* built-in routine call, then M is the mode denoted by *argument*.

A *tuple* is **constant** if and only if each *value* or *expression* occurring in it is **constant**.

static conditions: The optional *mode name* may be 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. For a constant powerset tuple the value delivered by each *expression* must be one of the values defined by that **member** mode.

b. *array tuple*

1. The mode of the *tuple* must be an array mode.

2. The class of each *value* must be **compatible** with the **element** mode of the mode of the *tuple*.

3. In the case of an *unlabelled array tuple*, there must be as many occurrences of *value* as the number of elements of the array mode of the *tuple*.

4. In the case of a *labelled array tuple*, the case selection conditions must hold for the list of *case label list* occurrences (see section 10.1.3). The **resulting class** of the list must be **compatible** with the **index** mode of the mode of the *tuple*.

5. In the case of a *labelled array tuple*, 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 be a value defined by the **index** mode of the *tuple*.

6. In an *unlabelled array tuple*, at least one *value* occurrence must be an *expression*.

7. For a **constant** (array) *tuple*, where the **element** mode of the mode of the *tuple* is a discrete mode, each specified *value* must deliver a value defined by that **element** mode, unless it is an **undefined** value.

8. In those contexts where the optional *mode name* can be deleted (as specified above), it is allowed to use a *character string literal* or a *bit string literal* provided that :

1. the mode of *tuple* is an array mode;
2. the **element** mode of the mode of *tuple* is **compatible** with the *CHAR(1)*- or *BIT(1)*-derived class if *tuple* is a *character string literal* or *bit string literal* respectively;

3. the **string length** of the *character string literal* or *bit string literal* must deliver the same value as the **number of elements** of the mode of *tuple*.

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 10.2.5).

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 any **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 any **field** name specified in the *field name list* labelling that *value*.

- If the mode of the *tuple* is a **tag-less variant** structure mode or a **tag-less 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*.

No *tuple* may have two *value* occurrences in it such that one is **extra-regional** and the other is **intra-regional** (see section 9.2.2).

dynamic conditions: The assignment conditions of any value with respect to the **member** mode, **element** mode or associated **field** mode, in the case of *powerset tuple*, *array tuple* or *structure tuple*, respectively (see section 6.2) apply (refer to conditions a2, b2, c4, c7, c10, c13 and c15).

If the *tuple* has a dynamic array mode, the *RANGEFAIL* exception occurs if any of the conditions b3 or b5 fail.

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

The value delivered by a *tuple* must not be **undefined**.

examples:

9.6	*number_list*[ ]	*(1.1)*
9.7	[*2:max*]	*(2.1)*
8.25	[*('A'):3,('B','K','Z'):1,(* **ELSE** *):0*]	*(6.1)*
17.5	[*(*):' '*]	*(6.1)*
12.35	*(: NULL , NULL ,536:)*	*(7.1)*
11.18	[*.status:occupied,.p:*[*white,rook*]]	*(9.1)*

5.2.6 Value string elements

syntax:

<value string element> ::= *(1)*
<string primitive value>(<start element>) *(1.1)*

derived syntax: A *value string element* is derived syntax for a *value string slice* of length 1 (see section 5.2.7), i.e. :
<string primitive value>(<start element>)
is derived from:
<string primitive value> (<start element> **UP** *1)*

5.2.7 Value string slices

syntax:

<value string slice> ::= *(1)*
<string primitive value>(<left element> : <right element>) *(1.1)*

| <*string primitive value*>(<*start element*> **UP** <*slice size*>) *(1.2)*

N.B. if the *string primitive value* is a *string location*, the syntactic construct is ambiguous and will be interpreted as a *string slice* (see section 4.2.7).

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

static properties: The (possibly dynamic) class of a *value string slice* is the M-value class, where M is a **parameterised** string mode constructed as :
&name (string size)
where *&name* is a virtual **synmode** name **synonymous** with the (possibly dynamic) mode of the *string primitive value* and where *string size* is either

NUM (right element) – NUM (left element) + 1

or

NUM (slice size).

The class of a *value string slice* is static if *string size* is **literal** : i.e., *left element* and *right element* are **literal** or *slice size* is **literal**; otherwise, the class is dynamic.

static conditions: If *left element* and *right element* are **literal**, or if *slice size* is **literal**, then they must deliver integer values such that the following relations hold:

$0 \leq NUM$ (*left element*) $\leq NUM$ (*right element*) $\leq L - 1$

$1 \leq NUM$ (*slice size*) $\leq L$

where L is the **string length** of the mode of the *string primitive value*. (If the mode of *string primitive value* is dynamic, these relations can only be checked at run time; see below.)

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

The *RANGEFAIL* exception occurs if any of the relations above does not hold in the case of a *string primitive value* which has a dynamic class, or if any of the following relations does not hold:

$0 \leq NUM$ (*left element*) $\leq NUM$ (*right element*) $\leq L - 1$

$0 \leq NUM$ (*start element*) $< NUM$ (*start element*) $+ NUM$ (*slice size*) $\leq L$

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

5.2.8 Value array elements

syntax:

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

N.B. if the *array primitive value* is an *array location* the syntactic construct is ambiguous and will be interpreted as an *array element* (see section 4.2.8).

derived syntax: See section 4.2.8

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 primitive value*.

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

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

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

$$L \leq \textit{expression} \leq U$$

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

5.2.9 Value array slices

syntax:

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

N.B. If the *array primitive value* is an *array location*, the syntactic construct is ambiguous and will be interpreted as an *array slice* (see section 4.2.9).

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

static properties: The (possibly dynamic) class of a *value array slice* is the M-value class, where M is a **parameterised** array mode constructed as:

&name(upper index)

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

NUM (upper index) = NUM (L) + NUM (upper element) - NUM (lower element)

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

NUM (upper index) = NUM (L) + NUM (slice size) - 1

where L is the **lower bound** of the mode of the *array primitive value*.

The class of a *value array slice* is static if *upper index* is **literal** : i.e., *lower element* and *upper element* both are **literal** or *slice size* is **literal**; otherwise, the class is dynamic.

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

If *lower element* and *upper element* both are **literal** or *slice size* is **literal**, then they must deliver values such that the following relations hold:

$$L \leq \textit{lower element} \leq \textit{upper element} \leq U$$

$$1 \leq \textit{NUM (slice size)} \leq \textit{NUM (U)} - \textit{NUM (L)} + 1$$

where L and U are, respectively, the **lower bound** and **upper bound** of the mode of the *array primitive value*. If the mode of *array primitive value* is dynamic, these relations can only be checked at run time; see below.

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

The *RANGEFAIL* exception occurs if any of the relations above does not hold for an *array primitive value* which has a dynamic class or if any of the following relations does not hold:

$L \leq$ *lower element* $\leq$ *upper element* $\leq U$

NUM (L) $\leq$ *NUM (first element)* $\leq$ *NUM (first element) + NUM (slice size) - 1* $\leq$ *NUM (U)*

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

5.2.10 Value structure fields

syntax:

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

N.B. if the *structure primitive value* is a *structure location* the syntactic construct is ambiguous and will be interpreted as a *structure field* (see section 4.2.10).

semantics: A value structure field delivers a value which is a field of the specified structure value. If the *structure primitive value* has a **tag-less variant** structure mode and the *field name* is a **variant** field name, the semantics are implementation defined.

static properties: The class of *value structure field* is the M-value class, where M is the mode of the *field name*.

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

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

The *TAGFAIL* exception occurs if the *structure primitive value* 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:

*16.51 (**RECEIVE** user_buffer).allocator* (1.1)

5.2.11 Expression conversions

syntax:

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

N.B. : if the *expression* is a *static mode location*, the syntactic construct is ambiguous and will be interpreted as a *location conversion* (see section 4.2.13).

semantics: An expression conversion overrides the CHILL mode checking and compatibility rules. It explicitly attaches a mode to the expression. If the mode of the mode name is a discrete mode and the class of the value delivered by the expression is discrete, then the value delivered by the expression conversion must be such that :

NUM (mode name (expression)) = NUM (expression)

Otherwise, the value delivered by the expression conversion is implementation defined and depends on the interal representation 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 *mode name* must not have the **non-value property**. An implementation may impose additional static conditions.

dynamic conditions: If the class of the value delivered by *expression* is discrete and if the mode of the *mode name* is a discrete mode which does not define a value with an internal representation equal to *NUM* (*expression*), then the *OVERFLOW* exception occurs. An implementation may impose additional dynamic conditions that, when violated, result in the occurrence of an exception defined by the implementation.

5.2.12 Value procedure calls

syntax:

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

semantics: A value procedure call delivers the value returned from a procedure.

static properties: The class of the *value procedure call* is the M-value class, where M is the mode of the **result spec** of the *value procedure call*.

dynamic conditions: The *value procedure call* must not deliver an **undefined** value (see sections 5.3.1 and 6.8).

examples:

```
6.50    julian_day_number([ 10,dec,1979])                    (1.1)
11.63   ok_bishop(b,m)                                       (1.1)
```

5.2.13 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>                  (1.2)

<CHILL value built-in routine call> ::=                      (2)
        NUM ( <discrete expression>)                         (2.1)
      | PRED ( <discrete expression> )                       (2.2)
      | SUCC ( <discrete expression> )                       (2.3)
      | ABS ( <integer expression> )                         (2.4)
      | CARD ( <powerset expression> )                       (2.5)
      | MAX ( <powerset expression> )                        (2.6)
      | MIN ( <powerset expression> )                        (2.7)
      | SIZE ( { <mode name> | <static mode location>} )     (2.8)
      | UPPER (<upper lower argument> )                      (2.9)
      | LOWER (<upper lower argument> )                      (2.10)
      | GETSTACK ( <getstack argument> [, <value> ])         (2.11)
      | ALLOCATE ( <allocate argument> [, <value> ])         (2.12)
      | <io CHILL value built-in routine call>               (2.13)

<getstack argument> ::=                                      (3)
        <argument>                                           (3.1)

<allocate argument> ::=                                      (4)
        <argument>                                           (4.1)

<argument> ::=                                               (5)
        <mode name>                                          (5.1)
      | <array mode name>( <expression> )                    (5.2)
      | <string mode name>( <integer expression> )           (5.3)
      | <variant structure mode name>( <expression list> )   (5.4)
```

<upper lower argument> ::= (6)
<array location> (6.1)
| <array primitive value> (6.2)
| <array mode name> (6.3)
| <string location> (6.4)
| <string primitive value> (6.5)
| <string mode name> (6.6)
| <discrete location> (6.7)
| <discrete expression> (6.8)
| <discrete mode name> (6.9)

N.B. : If the *upper lower argument* is an (*array*, *string*, *discrete*) *location*, the syntactic ambiguity is resolved by interpreting *upper lower argument* as a *location* rather than an *expression* or *primitive value*.

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 that delivers a value. The CHILL value built-in routines related to input output are defined in chapter 7.

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* deliver, respectively, the next lower and higher discrete value. If the discrete argument 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*).

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

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 and *LOWER* are defined on (possibly dynamic) :

- array, string and discrete locations, delivering the **upper bound** and **lower bound** of the mode of the location,
- array and string primitive values, delivering the **upper bound** and **lower bound** of the mode of the value's class,
- **strong** discrete expressions, delivering the **upper bound** and **lower bound** of the mode of the value's class,
- array, string and discrete mode names, delivering the **upper bound** and **lower bound** of the mode

respectively.

GETSTACK and *ALLOCATE* create a location of the specified mode and deliver a reference value for the created location. *GETSTACK* creates this location on the stack (see section 8.9). If the *argument* is a *mode name*, a static mode location of that mode is created and a reference value is delivered. Otherwise, a dynamic mode location is created whose mode is a **parameterised** mode with non-**literal** parameters as specified in the *argument* and a row value referring to the location

is delivered. The created location is initialised with the value of *value*, if present; otherwise, with the **undefined** value (see section 4.1.2).

static properties: The class of a *NUM* built-in routine call is the *INT* -derived class. The built-in routine call is **constant** if and only if the argument is either **constant** or **literal**.

The class of a *PRED* or *SUCC* built-in routine call is the **resulting class** of the argument. The built-in routine call is **constant** (**literal**) if and only if the argument is **constant** (**literal**).

The class of an *ABS* built-in routine call is the **resulting class** of the argument. The built-in routine call is **constant** (**literal**) if and only if the argument is **constant** (**literal**).

The class of a *CARD* built-in routine call is the *INT* -derived class. The built-in routine call is **constant** if and only if the argument is **constant**.

The class of a *MAX* or *MIN* built-in routine call is the M-value class, where M is the **member** mode of the mode of the *powerset expression*. The built-in routine call is **constant** if and only if the argument is **constant**.

The class of a *SIZE* built-in routine call is the *INT* -derived class. The built-in routine call is **constant**.

The class of an *UPPER* and *LOWER* built-in routine call is

- the M-value class if *upper lower argument* is an *array location*, *array expression* or *array mode name*, where M is the **index** mode of *array location*, (**strong**) *array expression* or *array mode name*, respectively;
- the *INT* -derived class if *upper lower argument* is a *string location*, *string expression* or *string mode name*;
- the M-value class if *upper lower argument* is a *discrete location*, *discrete expression* or *discrete mode name*, where M is the mode of *discrete location*, or the mode of the (**strong**) *discrete expression*, or the *discrete mode name* respectively.

An *UPPER* or *LOWER* built-in routine call is **constant** if the *upper lower argument* is an (*array*, *string* or *discrete*) *mode name* or if the mode of the *array* or *string location* is static or if the *array* or *string expression* has a static class or if *upper lower argument* is a *discrete expression* or a *discrete location*.

The class of a *GETSTACK* or *ALLOCATE* built-in routine call is the M-reference class, where M is the mode of *argument*. M is either the *mode name* or a **parameterised** mode constructed as :
&< 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 *static mode location* argument of *SIZE* must be **referable**.

The *discrete expression* as an argument of *UPPER* and *LOWER* must be **strong**.

The following compatibility requirements hold for an *argument* which is not a single *mode name*:

- The class of the *expression* must be **compatible** with the **index** mode of the *array mode name*.

- The *variant structure mode name* must be **parameterisable** and 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*.

The class of the *value*, if present, in the *GETSTACK* and *ALLOCATE* built-in routine call must be **compatible** with the mode of *argument*; this check is dynamic in case the mode of *argument* is a dynamic mode.

dynamic properties: A reference value is an **allocated** reference value if and only if it is returned by an *ALLOCATE* built-in routine call.

dynamic conditions: *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.

NUM and *CARD* cause the *OVERFLOW* exception if the resulting value is outside the set of values defined by *INT* .

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

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

GETSTACK and *ALLOCATE* cause the *RANGEFAIL* exception if in the *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 **string length** of the *string mode name*;
- any expression in the *expression list* for which the corresponding class in the list of classes of the *variant structure mode name* is an M-value class (i.e. is **strong**) delivers a value which is outside the set of values defined by M.

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

ALLOCATE causes the *ALLOCATEFAIL* exception if storage requirements cannot be satisfied.

For *GETSTACK* and *ALLOCATE* the assignment conditions of the value delivered by *value* with respect to the mode of *argument* apply.

examples:

9.12	*MIN (sieve)*	*(2.10)*
11.47	*PRED (col_1)*	*(2.2)*
11.47	*SUCC (col_1)*	*(2.4)*

5.2.14 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 9). 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 ($m \geq n$), the compatibility requirements for the first n actual parameters are the same as for procedure parameter passing (see section 6.7).

dynamic conditions: 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:

15.35 **START** *counter()* *(1.1)*

5.2.15 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.

5.2.16 Parenthesised expression

syntax:

<*parenthesised expression*> ::= *(1)*
(<*expression*>) *(1.1)*

semantics: A parenthesised expression delivers the value delivered by the evaluation of the expression.

static properties: The class of the *parenthesised expression* is the class of the *expression*.

A *parenthesised expression* is **constant** (**literal**) if and only if the *expression* is **constant** (**literal**).

examples:

5.10 *(a1 OR b1)* *(1.1)*

5.3 VALUES AND EXPRESSIONS

5.3.1 General

syntax:

<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 string slice*, a *value array element*, 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 context requires a **constant** or **literal** expression, the evaluation is assumed to be done prior to run time and cannot cause an exception. An implementation will define ranges of allowed values for literal and constant expressions and may reject a program if such a prior-to-run-time evaluation delivers a value out of the implementation defined bounds.

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:

10.31 i<min *(1.1)*
10.31 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:

5.10 (a1 OR b1) *(1.1)*
5.10 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 and they test whether or not a powerset value is contained in: <=, is properly contained in: <, contains: >= or properly contains: > the other powerset value. A powerset inclusion operator delivers a boolean value as result.

static properties: If an *operand-2* is an *operand-3*, the class of the *operand-2* is the class of the *operand-3*. If an *operator-3* is specified, the class of the *operand-2* is the *BOOL* -derived class.

An *operand-2* is **constant** (**literal**) if and only if it is either an *operand-3* which is **constant** (**literal**) or built up from a *sub operand-2* and an *operand-3* which are both **constant** (**literal**).

static conditions: If an *operator-3* is specified, the following compatibility requirements between the class of *sub operand-2* and the class of 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:

10.50 NULL *(1.1)*
10.50 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 dependent 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 **string lengths** of the **root** modes of both classes,
  - otherwise, the class is the *&name(n)*-value class, where *&name* is a virtual **synmode** name **synonymous** with the mode of one of the **strong** operands and *n* is the sum of the **string lengths** of the **root** modes of both classes.

  (this class is dynamic if one or both operands have a dynamic class).
- if *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**) and *operator-4* is either the *arithmetic additive operator* or the *powerset difference operator*.

static conditions: If an *operator-4* is specified, the following compatibility requirements must be fulfilled:

- if *operator-4* is the *arithmetic additive operator*, the classes of both operands must be **compatible** and they must both have an integer **root** mode;
- if *operator-4* is the *string concatenation operator*, the **root** modes of the classes of both operands must both be **compatible** with a **bit** string mode or both be **compatible** with a **character** string mode and, if both classes are value classes, their **root** modes must have the same **novelty**;
- if *operator-4* is the *powerset difference operator*, the classes of both operands must be **compatible** and both must have a powerset **root** mode.

dynamic conditions: In the case of an *operand-3* which is not **constant**, an *OVERFLOW* exception occurs if an addition (+) or a subtraction (-) gives rise to a value that is not one of the values defined by the **root** mode of the class of the *operand-3*.

examples:

1.6 j *(1.2)*

1.6 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 quotient operation is defined such that all relations
ABS (X/Y) = ABS (X) / ABS (Y) and
sign (X/Y) = sign (X) / sign (Y) and
*ABS (X) - (ABS (X) / ABS (Y)) * ABS (Y) = ABS (X) MOD ABS (Y)*
yield *TRUE* for all integer values X and Y, where *sign (X) = -1* if $X < 0$, otherwise *sign (X) = 1*.

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*, that is not **constant**, an *OVERFLOW* exception occurs if a multiplication (*) or a division (/) or a modulo (*MOD*) or a remainder (*REM*) operation gives rise to a value that is not one of the values defined by the **root** mode of the class of 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, is delivered.

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 result is 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 **string length** of the string literal.

An *operand-5* is **constant** if and only if the *operand-6* is **constant**.
An *operand-5* is **literal** if and only if the *operand-6* is **literal** and the *monadic operator* is – or *NOT* .

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 one of the values defined by the root mode of the class of the operand-5.

examples:

5.10	*NOT k2*	*(1.1)*
7.54	*(6)' '*	*(1.1)*
7.54	*(6)*	*(2.2)*

5.3.8 Operand-6

syntax:

<operand-6> ::=	*(1)*
<referenced location>	*(1.1)*
| *<receive expression>*	*(1.2)*
| *<primitive value>*	*(1.3)*
<referenced location> ::=	*(2)*
–> <location>	*(2.1)*
| **ADDR** *(<location>)*	*(2.2)*
<receive expression> ::=	*(3)*
RECEIVE *<buffer location>*	*(3.1)*

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

semantics: An operand-6 is either a referenced location, a receive expression or a primitive value (see section 5.2.1).

A referenced location delivers a reference to the specified location.

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 9 for full details).

static properties: The class of an *operand-6* is the class of the *referenced location*, *receive expression* or *primitive value* respectively.
The class of the *referenced location* is the M-reference class where M is the mode of the *location*.
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*.

An *operand-6* is **constant** if and only if the *primitive value* is **constant** or the *referenced location* is **constant**.
A *referenced location* is **constant** if and only if the *location* is **static**.
An *operand-6* is **literal** if and only if the *primitive value* is **literal**.

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

dynamic conditions: The lifetime of the *buffer location* must not end while the executing process is delayed on that buffer location.

examples:

8.24	*-> c*	*(2.1)*
16.51	**RECEIVE** *user_buffer*	*(3.1)*

6 ACTIONS

6.1 GENERAL

syntax:

<*action statement*> ::= (1)
[<*defining occurrence*> :] <*action*> [<*handler*>] [<*simple name string*>]; (1.1)
| <*module*> (1.2)
| <*spec module*> (1.3)

<*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)
| <*begin-end block*> (3.4)
| <*delay case action*> (3.5)
| <*receive case action*> (3.6)

semantics: Action statements constitute the algorithmic part of a CHILL program. Any action statement may be labelled. Those actions that may never cause an exception may never have a handler appended.

static properties: A *defining occurrence* in an *action statement* defines a **label** name.

static conditions: The *simple name string* may only be given after an *action* which is a *bracketed action* or if a *handler* is specified, and only if a *defining occurrence* is specified. The *simple name string* must be the same name string as the *defining occurrence.*

6.2 ASSIGNMENT ACTION

syntax:

<*assignment action*> ::= (1)
<*single assignment action*> (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 of 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 semantics for the variant fields that depend on it are implementation defined.

static conditions: The modes of all *location* occurrences must be **equivalent** and they must have neither the **read-only property**, nor the **non-value property**. Each mode must be **compatible** with the class of the *value*. The checks are dynamic in the case where dynamic mode locations and/or a value with a dynamic class are involved.

The *value* must be **regionally safe** for every *location* (see section 9.2.2).

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 a range mode and the value delivered by the evaluation of *value* is neither one of the values defined by the range mode nor the *undefined value*.

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 *assigning operator* the same exceptions are caused as if the expression:
<*location*> <*closed dyadic operator*> (<*expression*>)
were evaluated and the delivered value stored into the specified location (note that the location is evaluated once only).

examples:

4.12 a := b+c *(1.1)*
10.25 stackindex– := 1 *(2.1)*
19.19 x.prex, x.next := NULL *(3.1)*
10.25 – := *(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, is entered.

examples:

```
7.22    IF n >= 50 THEN rn(r) := 'L';
            n- := 50;
            r+ := 1;
        FI                                              (1.1)
10.50   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 list 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 10.1.3).

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 by a *discrete mode* in a *case label* (see section 10.1.3) 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 **reserved** simple name string **ELSE** , followed by an *action statement list*, may only be omitted if the list of *case label list* occurrences is **complete** (see section 10.1.3).

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:

```
4.11    CASE order OF
            (1): a := b+c;
                 RETURN ;
            (2): d := 0;
            ( ELSE ): d := 1;
        ESAC                                                    (1.1)
11.43   starting.p.kind, starting.p.color                       (2.1)
11.58   (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:

```
4.17    DO FOR i := 1 TO c;
            op(a,b,d,order-1);
            d := a;
        OD                                                    (1.1)
15.58   DO WITH each;
            IF this_counter = counter
                THEN
                 status := idle;
                  EXIT find_counter;
            FI ;
        OD                                                    (1.1)
```

6.5.2 For control

syntax:

<*for control*> ::= (1)
FOR { <*iteration*> { ,<*iteration*>}* | **EVER** } (1.1)

<*iteration*> ::= (2)
<*value enumeration*> (2.1)
| <*location enumeration*> (2.2)

<*value enumeration*> ::= (3)
<*step enumeration*> (3.1)
| <*range enumeration*> (3.2)
| <*powerset enumeration*> (3.3)

<*step enumeration*> ::= (4)
<*loop counter*> <*assignment symbol*>
<*start value*> [<*step value*>] [**DOWN**] <*end value*> (4.1)

<*loop counter*> ::= (5)
<*defining occurrence*> (5.1)

<*start value*> ::= (6)
<*discrete expression*> (6.1)

<*step value*> ::= (7)
BY <*integer expression*> (7.1)

<*end value*> ::= (8)
TO <*discrete expression*> (8.1)

<*range enumeration*> ::= (9)
<*loop counter*> [**DOWN**] **IN** <*discrete mode*> (9.1)

<*powerset enumeration*> ::= (10)
<*loop counter*> [**DOWN**] **IN** <*powerset expression*> (10.1)

<*location enumeration*> ::= (11)
<*loop counter*> [**DOWN**] **IN** <*composite location*> (11.1)

<*composite location*> ::= (12)
<*array location*> (12.1)
| <*string location*> (12.1)

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* , or 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, or if the handler of the do action is entered and falls through, or if the do action is left by a return 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 always implicitly defines a name which denotes its value or location inside the action statement list. However, if an access name with a name string that is equal to the name string 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)).

   Normal termination occurs 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, the next value will be the next greater (smaller) member value in the powerset value. Normal termination occurs if the action statement list has been executed for the greatest (smallest) value. When the do action is executed, the **powerset** expression is evaluated only once.

   step enumeration:

In the case of step enumeration without (with) **DOWN** specification, the set of values of the loop counter is determined by a start value, end value, and possibly step value. When the do action is executed, these expressions are evaluated only once in an unspecified and 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)).

Normal termination occurs if the evaluation yields a value which is greater (smaller) than the end value or would have caused an *OVERFLOW* exception.

3. **location enumeration**:

In the case of a location enumeration without (with) **DOWN** specification, the action statement list is repeatedly entered for a set of locations which are the elements of the array location denoted by *array location* or the components of the string location denoted by *string location*. The semantics are as if before each execution of the action statement list the loc-identity declaration :

DCL *<loop counter>* *<mode>* **LOC** := *<composite location>* *(<index>)*;

were encountered, where *mode* is the element mode of the *array location* or *&name(1)* such that *&name* is a virtual **synmode** name **synonymous** with the mode of the *string location*, and where *index* is initially set to the **lower bound** (**upper bound**) of the mode of *array location* or *string location* and *index* before each subsequent execution of the action statement list is set to *SUCC (index) (PRED (index))*. The action statement list will not be executed if the **string length** of the mode of *string location* = 0.
The do action is terminated (normal termination) if *index* just after an execution of the action statement list is equal to the **upper bound** (**lower bound**) of the mode of *array location* or *string location*.
When the do action is executed, the *composite location* is evaluated only once.

static properties: A loop counter has a name string attached which is the name string of its defining occurrence.

value enumeration:

The name defined by the *loop counter* is a **value enumeration** name. If a name string is visible in the reach in which the *do action* is placed which is equal to the name string of the *loop counter*, the *loop counter* is **explicit**, otherwise it is **implicit**.

step enumeration:

The class of the name defined by 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 the name defined by 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 name defined by the *loop counter* is the M-value class, where M is the *discrete mode*.

powerset enumeration:

The class of the name defined by the *loop counter* is the M-value class, where M is the **member** mode of the mode of the (**strong**) *powerset expression*.

location enumeration:

The name defined by the *loop counter* is a **location enumeration** name. Its mode is the **element** mode of the mode of the *array location* or the string mode *&name(1)*, where *&name* is a virtual **synmode** name **synonymous** with the mode of *string location*.

A **location enumeration** name is **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 name defined by the *loop counter*.

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:

4.17	**FOR** *i* := *1* **TO** *c*	*(1.1)*
15.37	**FOR EVER**	*(1.1)*
4.17	*i* := *1* **TO** *c*	*(3.1)*
9.12	*j* := *MIN (sieve)* **BY** *MIN (sieve)* **TO** *max*	*(3.1)*
14.28	*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:

7.35 **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 primitive value>* *(2.2)*

N.B. if the *structure primitive value* is a *location*, the syntactic construct is ambiguous and will be interpreted as a *structure location*.

semantics: The (visible) field names of the mode of the structure locations or structure value specified in each *with control* are made available as direct accesses to the fields.

The visibility rules are as if a field name defining occurrence were introduced for each field name attached to the mode of the location or primitive value and with the same name string as the field name.

If a *structure location* is specified, access names with the same name string as the field names of the mode of the *structure location* are implicitly defined, denoting the sub-locations of the structure location.

If a *structure primitive value* is specified, value names with the same name string as the field names of the mode of the (**strong**) *structure primitive value* are implicitly defined, denoting the sub-values of the structure value.

When the do action is entered, the specified structure locations and/or structure values are evaluated once only on entering the do action, in an unspecified, possibly mixed order.

static properties: The (virtual) defining occurrence introduced for a **field** name has the same name string as the *field name defining occurrence* of that **field** name.

Structure primitive value: A (virtual) defining occurrence in a *with-part* defines a **value do-with** name. Its class is the M-value class, where M is the mode of that **field** name of the structure mode of the *structure primitive value*, which is made available as **value do-with** name.

Structure location: A (virtual) defining occurrence in a *with-part* defines a **location do-with** name. Its mode is the mode of that **field** name of the mode of the *structure location*, which is made available as **location do-with** name. A **location do-with** name is **referable** if the field layout of the associated **field** name is **NOPACK** .

examples:

15.58 **WITH** *each* *(1.1)*

6.6 EXIT ACTION

syntax:

<exit action> ::= *(1)*
EXIT *<simple name string>* *(1.1)*

semantics: An exit action is used to leave a bracketed action statement or a module. Action is resumed immediately after the closest surrounding bracketed action statement or module labelled with the simple name string.

static conditions: The *exit action* must lie within the bracketed action statement or module of which the *defining occurrence* in front has the same name string as *simple name string*.

If the *exit action* is placed within a procedure or process definition, the exited bracketed action statement or module must also lie within the same procedure or process definition (i.e. the exit action cannot be used to leave procedures or processes).

No *handler* may be appended to an *exit action*.

examples:

15.62 **EXIT** *find_counter* *(1.1)*

6.7 CALL ACTION

syntax:

```
<call action> ::=                                                    (1)
        [ CALL ] { <procedure call>                                  (1.1)
      | <CHILL built-in routine call>                                (1.2)
      | <implementation built-in routine call>}                      (1.3)

<procedure call> ::=                                                 (2)
        { <procedure name> | <procedure primitive value>} ( [ <actual parameter list> ] )(2.1)

<actual parameter list> ::=                                          (3)
        <actual parameter> { ,<actual parameter>} *                  (3.1)

<actual parameter> ::=                                               (4)
        <value>                                                      (4.1)
      | <location>                                                   (4.2)

<CHILL built-in routine call> ::=                                    (5)
        <CHILL value built-in routine call>                          (5.1)
      | <CHILL location built-in routine call>                       (5.2)
      | <CHILL simple built-in routine call>                         (5.3)

<CHILL simple built-in routine call> ::=                             (6)
        TERMINATE (<reference expression>)                           (6.1)
      | <io CHILL simple built-in routine call>                      (6.2)
```

derived syntax: The **reserved** simple name string **CALL** is optional. A *call action* with **CALL** is derived from a *call action* without **CALL** .

semantics: A call action causes either the call of a procedure or of a built-in routine. A procedure call causes a call of the general procedure indicated by the value delivered by the procedure primitive value or the procedure indicated by the procedure name. The actual values and locations specified in the actual parameter list are passed to the procedure.

A CHILL built-in routine call is either a CHILL location built-in routine call, which delivers a location (see section 4.2.12), or a CHILL value built-in routine call, which delivers a value (see section 5.2.13), or a CHILL simple built-in routine call, which delivers neither a value nor a location. The simple built-in routines for input output are described in Chapter 7.

TERMINATE ends the lifetime of the location referred to by the value delivered by *reference expression*. An implementation might as a consequence release the storage occupied by this location. If the lifetime of the location had already ended prior to calling *TERMINATE* , no action is performed.

static properties: A *procedure call* has the following properties attached: a list of **parameter specs**, possibly a **result spec**, a possibly empty set of exception names, a **generality**, a **recursivity**, and possibly it is **intra-regional** (the latter is only possible with a *procedure name*, see section 9.2.2). These properties are inherited from the *procedure name* or any mode **compatible** with the class of the *procedure primitive value* (in the latter case, the generality is always **general**).

A *procedure call* with a **result spec** is a *location procedure call* if and only if **LOC** is specified in the **result spec**; otherwise, it is a *value procedure call*.

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 parameter spec has the **IN** attribute (default), the *actual parameter* must be a *value* whose class is **compatible** with the mode in the corresponding parameter spec. The latter mode must not have the **non-value property**. The *actual parameter* is a *value* which must be **regionally safe** for the *procedure call*.

- If the parameter spec has the **INOUT** or **OUT** attribute, the *actual parameter* must be a *location*, whose mode must be **compatible** with the M-value class, where M is the mode in the corresponding parameter spec. The mode of the (actual) *location* must be static and must not have the **read-only property** nor the **non-value property**. The *actual parameter* is a *location*. It can be viewed as a *value* which must be **regionally safe** for the *procedure call*.

- If the parameter spec has the **INOUT** attribute, the mode in the parameter spec must be **compatible** with the M-value class where M is the mode of the *location*.

- If the parameter spec has the **LOC** attribute specified without **DYNAMIC** , the *actual parameter* must be a *location* which is both **referable** and such that the mode in the parameter spec is **read-compatible** with the mode of the (actual) *location*, or the *actual parameter* must be a *value* which is not a *location* but whose class is **compatible** with the mode in the parameter spec.

- If the parameter spec has the **LOC** attribute with **DYNAMIC** specified, the *actual parameter* must be a *location* which is both **referable** and such that the mode in the parameter spec is **dynamic read-compatible** with the mode of the (actual) *location*, or the *actual parameter* must be a *value* which is not a *location* but whose class is **compatible** with a parameterised version of this mode.

- If the parameter spec has the **LOC** attribute then

  - if the *actual parameter* is a *location* it must have the same **regionality** as the *procedure call*;

  - if the *actual parameter* is a *value* then it must be **regionally safe** for the *procedure call*.

dynamic conditions: A procedure call can cause any of the exceptions of the attached set of exception names. It causes the *EMPTY* exception if the *procedure primitive value* 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 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 primitive value* must not deliver a procedure defined within a process definition whose activation is not the same as the activation of the process executing the procedure call (other than the imaginary outermost process) and the lifetime of the denoted procedure must not have ended.

TERMINATE causes the *EMPTY* exception if the *reference expression* delivers the value *NULL* .

TERMINATE causes the *TERMINATEFAIL* exception if the *reference expression* does not deliver an **allocated** reference value.

examples:

4.18 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)*
| *<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** (**LOC DYNAMIC**)is specified in the **result spec** of the **procedure** name of the *result action*, the *result* must be a *location*, such that the mode in the **result spec** is **read-compatible** (**dynamic read-compatible**) with the mode of the *location*. The *location* must be **referable** if **NONREF** is not specified in the **result spec**. The *result* is a *location* which must have the same **regionality** as the *procedure name* attached to the *result action*.

If **LOC** is not specified in the **result spec** of the **procedure** name of the *result action*, the *result* must be a *value*, whose class is **compatible** with the mode in the **result spec**. The *result* is a *value* which must be **regionally safe** for the *procedure name* attached to the *result action*.

dynamic conditions: If **LOC** is not specified in the **result spec** of the **procedure** name, the assignment conditions of the *value* in the *result action*, with respect to the mode in the **result spec** of its **procedure** name, apply.

examples:

4.21 **RETURN** *(1.1)*
1.6 **RESULT** *i+j* *(2.1)*
5.19 *c* *(3.1)*

6.9 GOTO ACTION

syntax:

<goto action> ::= *(1)*
GOTO *<simple name string>* *(1.1)*

semantics: The goto action causes a transfer of control. Action is resumed with the action statement labelled with the simple name string.

static conditions: If the *goto action* is placed within a procedure or process definition, the label indicated by the *simple name string* 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:

4.7 **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 the exception whose name is indicated by *exception name*.

static conditions: A *handler* must not be appended to a *cause action*.

examples:

4.9 **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 5.2.14), possibly without using the resulting instance value.

examples:

14.45 **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 9.1).

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 re-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 9 for further details).

examples:

13.25 **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 re-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 9 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:

13.18 **DELAY** *resource_freed* *(1.1)*

6.17 DELAY CASE ACTION

syntax:

<*delay case action*> ::= (1)
DELAY CASE [{ **SET** <*instance location*> [<*priority*>] ; | <*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 re-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 9 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:

14.26
```
DELAY CASE
   (operator_is_ready): /* some actions */
   (switch_is_closed): DO FOR i IN INT (1:100);
                          CONTINUE operator_is_ready;
                       /* empty the queue */
                       OD ;
ESAC                                                    (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 primitive value>*] [*<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 signal. This process identification must not be in contradiction with a possible process name attached to the signal name. If neither an *instance primitive value* is specified nor a process name is attached to the signal name, the signal may be received by any process.

static conditions: The number of *value* occurrences must be equal to the number of modes of the *signal name*. The class of each *value* must be **compatible** with the corresponding mode of the *signal name*. No *value* occurrence may be **intra-regional** (see section 9.2.2). The *integer literal expression* in *priority* must not deliver a negative value.

dynamic conditions: The assignment conditions of each *value*, with respect to its corresponding mode of the *signal name*, apply.

The *EMPTY* exception occurs if the *instance primitive value* delivers *NULL* .

The *EXTINCT* exception occurs if and only if the lifetime of the process indicated by the value delivered by the *instance primitive value* has terminated at the point of the execution of the send signal action.

The *SENDFAIL* exception occurs if the *signal name* has a **process** name attached which is not the name of the process indicated by the value delivered by the *instance primitive value*.

examples:

15.78 **SEND** *ready* **TO** *received_user* *(1.1)*

15.86 **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 9 for further details).

static conditions: The class of the *value* must be **compatible** with the **buffer element mode** mode of the mode of the *buffer location*. The *value* must not be **intra-regional** (see section 9.2.2). 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:

16.119 **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 9 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** *<defining occurrence 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 the *instance location* is evaluated 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.

When a signal is 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 defining occurrences must be specified after **IN** .
They define **value receive** 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 and if the signal is received, the **instance** value denoting the process that has sent the received signal will be stored into the *instance location* immediately after signal reception and before entering the *signal receive alternative*.

static properties: A *defining occurrence* in the *defining occurrence list* of a *signal receive alternative* defines a **value receive** name. Its class is the M-value class, where M is the corresponding mode 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 *defining occurrence list* in the *signal receive alternative* must be specified if and only if the *signal name* has a non-empty set of modes. The number of names in the *defining occurrence list* must be equal to the number of modes of the *signal name*.

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**; if it has the **referencing property** then the *access name* must be **extra-regional**.

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 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:

```
15.83   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** *<defining occurrence>*) : *<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 the *instance location* is evaluated 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 possibly 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 defining occurrence after **IN** defines a **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 and if the value is received, the **instance** value denoting the process that has sent the received value will be stored into the *instance location* immediately after reception of the value and before entering the *buffer receive alternative*.

static properties: A *defining occurrence* in a *buffer receive alternative* defines a **value receive** name. Its class is the M-value class, where M is the **buffer element** mode of the mode of the *buffer location* labelling the *buffer receive alternative*.

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. The mode of the *access name* must not have the **read-only property**; if it has the **referencing property** then the *access name* must be **extra-regional**.

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.

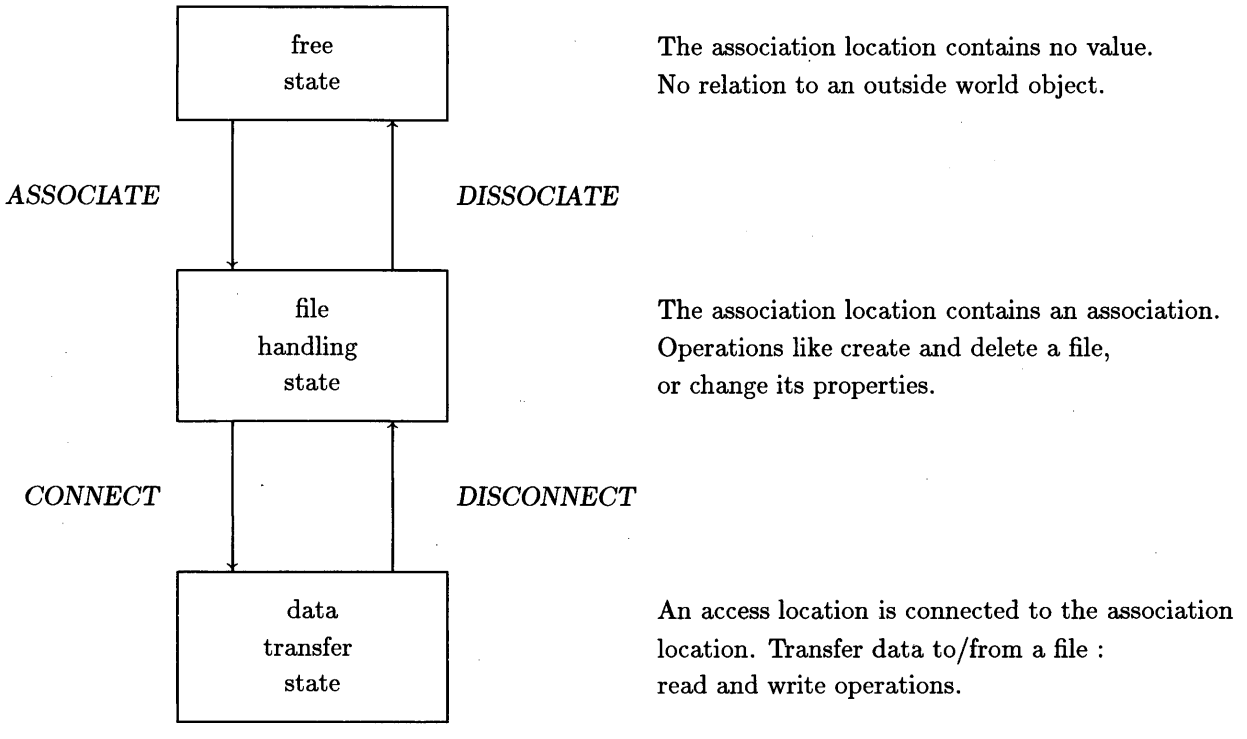
The lifetime of none of the delivered **buffer** locations must end while the process executing the receive buffer case action is delayed on it.

7 INPUT AND OUTPUT

7.1 I/O REFERENCE MODEL

A model is used for the description of the input/output facilities in an implementation independent way; it distinguishes 3 states for a given association location : a free state, a file handling state and a data transfer state.

The diagram shows the three states and the possible transitions between the states.



The model assumes that objects, in implementations often referred to as datasets, files or devices, exist in the **outside world**, i.e., the external environment of a CHILL program. Such an outside world object is called a **file** in the model. A file can be a physical device, a communication line or just a file in a file management system; in general, a file is an object that can produce and/or consume data.

Manipulating a file in CHILL requires an **association**; an association is created by the associate operation and it identifies a file. An association has **attributes**; these attributes describe the properties of a file that is or could be attached to the association.

In the **free state**, there is no interaction or relation between the CHILL program and outside world objects. The associate operation changes the state of the model from the free state into the **file handling state**. This operation takes as one argument an association location and an implementation defined denotation for an outside world object for which an association must be created; additional arguments may be used to indicate the kind of association for the object and the initial values for the attributes of the association. A particular association also implies an (implementation dependent) set of operations that may be applied on the file that is attached to that association.

In the file handling state, it is possible to manipulate a file and its properties via an association, provided that the association enables the particular operation; for operations that change the properties of a file, an exclusive association for the file will be necessary in general.

The model assumes associations in general are exclusive : i.e., only one association exists at the same time for a given outside world object. However, implementations may allow the creation of more associations for the same object, provided that the object can be shared among different users (programs) and/or among different

associations within the same program. All operations in the file handling state take an association as an argument.

The **dissociate** operation is used to end an association for an outside world object; this operation causes transition from the file handling state back to the free state.

Transferring data to or from a file is possible only in the **data transfer state**; transfer operations require an **access** location to be **connected** to an association for that file. The connect operation connects an access location to an association and changes the state of the model into the data transfer state. The operation takes an association location and an access location as arguments; the association location contains an association for the file to, or from, which data can be transferred via the access location. Additional arguments of the connect operation denote for which type of transfer operations the access location must be connected, and to which record the file must be positioned. At most one access location can be connected to an association location at any one time.

The **disconnect** operation takes an access location as argument and disconnects it from the association it is connected to; it changes the state of the model back to the file handling state.

In the data transfer state, an access location must be used as an argument of a transfer operation; there are two transfer operations provided, namely, a **read** operation to transfer data from a file to the program and a **write** operation to transfer data from the program to a file. The transfer operations use the record mode of the access location to transform CHILL values into records of the file, and vice versa.

A file is viewed in the model as an **array of values**; each element of this array relates to a record of the file. The element mode of this array is determined by the connect operation to be the record mode of the access location being connected. An index value is assigned to each record of the file; this value uniquely identifies each record of the file. In the description of the connect and transfer operations, three special index values will be used, namely, a **base** index, a **current** index and a **transfer** index. The base index is set by the connect operation and remains unchanged until a subsequent connect operation; it is used to calculate the transfer index in transfer operations and the current index in a connect operation. The transfer index denotes the position in the file where a transfer will take place; the current index denotes the record to which the file currently is positioned.

7.2 ASSOCIATION VALUES

7.2.1 General

An association value reflects the properties of a file that is or could be attached to it. A particular association value also implies an (implementation dependent) set of operations on the file that is possibly attached to it.

Association values have no denotation but are contained in locations of assocation mode; there exists no expression denoting a value of association mode. Association values can only be manipulated by built-in routines that take an association location as parameter.

7.2.2 Attributes of association values

An association value has attributes; the attributes describe the properties of the association and the file that may or could be attached to it.

The following attributes are language defined :

- **existing** : indicating that a (possibly empty) file is attached to the association;
- **readable** : indicating that read operations are possible for the file when it is attached to the association;
- **writeable** : indicating that write operations are possible for the file when it is attached to the association;
- **indexable** : indicating that the file, when it is attached to the association, allows for random access to its records;

- **sequencible** : indicating that the file, when it is attached to the association, allows for sequential access to its records;

- **varying** : indicating that the size of the records of the file, when it is attached to the association, may vary within the file.

These attributes have a boolean value; the attributes are initialized when the association is created and may be updated as a consequence of particular operations on the association. This list comprises the language defined attributes only; implementations may add attributes according to their own needs.

7.3 ACCESS VALUES

7.3.1 General

Access values are contained in locations of access mode. An access location is necessary to transfer data from or to a file in the outside world.

Access values have no denotation but are contained in locations of access mode; there exists no expression denoting a value of access mode. Access values can only be manipulated by built-in routines that take an access location as parameter.

7.3.2 Attributes of access values

Access values have attributes that describe their dynamic properties, the semantics of transfer operations, and the conditions under which exceptions can occur.

CHILL defines the following attributes :

- **usage** : indicating for which transfer operation(s) the access location is connected to an association; the attribute is set by the connect operation.

- **outoffile** : indicating whether or not the transfer index calculated by the last read operation was in the file; the attribute is initialized to *FALSE* by the connect operation and is set by every read operation.

7.4 BUILT-IN ROUTINES FOR INPUT OUTPUT

7.4.1 General

Language defined built-in routines are defined for operations on association locations and access locations, and for inspecting and changing the attributes of their values.

syntax:

<io CHILL value built-in routine call> ::= *(1)*
<association attr io CHILL value built-in routine call> *(1.1)*
| *<isassociated io CHILL value built-in routine call>* *(1.2)*
| *<access attr io CHILL value built-in routine call>* *(1.3)*
| *<readrecord io CHILL value built-in routine call>* *(1.4)*

<io CHILL simple built-in routine call> ::= *(2)*
<dissociate io CHILL simple built-in routine call> *(2.1)*
| *<modification io CHILL simple built-in routine call>* *(2.2)*
| *<connect io CHILL simple built-in routine call>* *(2.3)*
| *<disconnect io CHILL simple built-in routine call>* *(2.4)*
| *<writerecord io CHILL simple built-in routine call>* *(2.5)*

<io CHILL location built-in routine call> ::= *(3)*
<associate io CHILL location built-in routine call> *(3.1)*

The built-in routines will be described in the following sections.

7.4.2 Associating an outside world object

syntax:

<associate io CHILL location built-in routine call> ::= *(1)*
ASSOCIATE (<association location>[,<associate parameter list>]) *(1.1)*

<isassociated io CHILL value built-in routine call> ::= *(2)*
ISASSOCIATED (<association location>) *(2.1)*

<associate parameter list> ::= *(3)*
<associate parameter> { ,*<associate parameter>* } * *(3.1)*

<associate parameter> ::= *(4)*
<location> *(4.1)*
| *<value>* *(4.2)*

semantics: *ASSOCIATE* creates an association to an outside world object. It initializes the *association location* with the created association. It initializes the attributes of the created association. The association location is also returned as a result of the call. The particular association that is created is determined by the locations and/or values occurring in the *associate parameter list*; the modes (classes) and the semantics of these locations (values) are implementation defined.

ISASSOCIATED returns *TRUE* if *association location* contains an association and *FALSE* otherwise.

static properties: The class of an *ISASSOCIATED* built-in routine call is the *BOOL* -derived class. The mode of an *ASSOCIATE* built-in routine call is the mode of the *association location*.

static conditions: The mode and the class of each *associate parameter* is implementation defined.

dynamic conditions: *ASSOCIATE* causes the *ASSOCIATEFAIL* exception if the *association location* already contains an association or if the association cannot be created due to implementation defined reasons.

example:

20.21 ASSOCIATE (file_association,'DSK:RECORDS.DAT'); *(1.1)*

7.4.3 Dissociating an outside world object

syntax:

<dissociate io CHILL simple built-in routine call> ::= *(1)*
DISSOCIATE (<association location>) *(1.1)*

semantics: *DISSOCIATE* terminates an association to an outside world object. An access location that is still connected to the association contained in an association location is disconnected before the association is terminated.

dynamic conditions: *DISSOCIATE* causes the *NOTASSOCIATED* exception if *association location* does not contain an association.

example:

22.38 DISSOCIATE (association); *(1.1)*

7.4.4 Accessing association attributes

syntax:

<*association attr io CHILL value built-in routine call*> ::= (1)
EXISTING (<association location>) (1.1)
| READABLE (<association location>) (1.2)
| WRITEABLE (<association location>) (1.3)
| INDEXABLE (<association location>) (1.4)
| SEQUENCIBLE (<association location>) (1.5)
| VARYING (<association location>) (1.6)

semantics: *EXISTING* , *READABLE* , *WRITEABLE* , *INDEXABLE* , *SEQUENCIBLE* and *VARYING* return respectively the value of the **existing**-, **readable**-, **writeable**-, **indexable**-, **sequencible**- and **varying**-attribute of the association contained in *association location*.

static properties: The class of an *association attr io CHILL value built-in routine call* is the *BOOL* -derived class.

dynamic conditions: The *association attr io CHILL value built-in routine call* causes the *NOTASSOCIATED* exception if *association location* does not contain an association.

7.4.5 Modifying association attributes

syntax:

<*modification io CHILL simple built-in routine call*> ::= (1)
CREATE (<association location>) (1.1)
| DELETE (<association location>) (1.2)
| MODIFY (<association location>[,<*modify parameter list*>]) (1.3)

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

<*modify parameter*> ::= (3)
<*value*> (3.1)
| <*location*> (3.2)

semantics: *CREATE* creates an empty file and attaches it to the association denoted by the *association location*. The **existing**-attribute of the indicated association is set to *TRUE* if the operation succeeds.

DELETE detaches a file from the association denoted by *association location* and deletes the file. The **existing**-attribute of the indicated association is set to *FALSE* if the operation succeeds.

MODIFY provides the means of changing properties of an outside world object for which an association exists and that is denoted by *association location*; the locations and/or values that occur in *modify parameter list* describe how the properties must be modified. The modes (classes) and the semantics of these locations (values) are implementation defined.

dynamic conditions: *CREATE* , *DELETE* and *MODIFY* cause the *NOTASSOCIATED* exception if *association location* does not contain an association.

CREATE causes the *CREATEFAIL* exception if one of the following conditions occurs :

- the **existing**-attribute of the association is *TRUE*;
- the creation of the file fails (implementation defined).

DELETE causes the *DELETEFAIL* exception if one of the following conditions occurs :

- the **existing**-attribute of the association is *FALSE*;
- the deletion of the file fails (implementation defined).

MODIFY causes the *MODIFYFAIL* -exception if the properties, defined by *modify parameter list* cannot or may not be modified; the conditions under which this exception can occur are implementation defined.

example:

```
21.39   CREATE (outassoc);                                   (1.1)
21.69   DELETE (curassoc);                                   (1.2)
```

7.4.6 Connecting an access location

syntax:

```
<connect io CHILL simple built-in routine call> ::=                          (1)
        CONNECT (<access location>,<association location>,
        <usage expression>[,<where expression>[,<index expression> ]])       (1.1)

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

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

<index expression> ::=                                                       (4)
        <expression>                                                         (4.1)
```

predefined set-modes: To control the connect operation, performed by the built-in routine *CONNECT* , two setmode names are predefined in the language, namely, *USAGE* and *WHERE*; the corresponding setmodes are **SET** (*READONLY* , *WRITEONLY* , *READWRITE*) and **SET** (*FIRST* , *SAME* , *LAST*), respectively.

Values of the mode *USAGE* indicate for which type of transfer operations the access location must be connected to an association, while values of the mode *WHERE* indicate how the file that is attached to an association must be positioned by the connect operation.

semantics: *CONNECT* connects the *access location* to the association that is contained in *association location*; there must be a file attached to the denoted association; i.e., the association's **existing**-attribute must be *TRUE*.

The value that is delivered by *usage expression* indicates for which type of transfer operations the access location must be connected to the file. If the expression delivers *READONLY* , the connection is prepared for read operations only; if it delivers *WRITEONLY* , the connection is set up for write operation only; if it delivers *READWRITE* the connection is prepared for both read and write operations.

The **indexable**-attribute of the denoted association must be *TRUE* if the access location has an index mode, while the **sequencible**-attribute must be *TRUE* if the location has no index mode.

CONNECT (re)positions the file that is attached to the denoted association; i.e., it establishes a (new) **base** index and **current** index in the file. The (new) **base** index depends upon the value that is delivered by *where expression* :

- if *where expression* delivers *FIRST* or is not specified, the **base** index is set to 0; i.e., the file is positioned before the first record;
- if *where expression* delivers *SAME* , the **base** index is set to the **current** index in the file; i.e., the file position is not changed;

- if *where expression* delivers *LAST* , the **base** index is set to N, where N denotes the number of records in the file; i.e., the file is positioned after the last record.

After a **base** index is set, a **current** index will be established by *CONNECT* . This **current** index depends upon the optional specification of an *index expression* :

- if no *index expression* is specified, the **current** index is set to the (new) **base** index;
- if an *index expression* is specified, the current index is set to
  base index + *NUM (v)* – *NUM (l)*
  where *l* denotes the **lower bound** of the access location's index mode and *v* denotes the value that is delivered by *index expression.*

If the access location is being connected for sequential write operations (i.e., the access location has no index mode and the *usage expression* delivers *WRITEONLY*), then those records in the file that have an index greater than the (new) **current** index will be removed from the file; i.e., the file may be truncated or emptied by *CONNECT* .

An access location that has no index mode cannot be connected to an association for read and write operations at the same time.

Any access location to which the denoted association may be connected will be disconnected implicitly before the association is connected to the location that is denoted by *access location.*

CONNECT initializes the **outoffile**-attribute of the access location to *FALSE* and sets the **usage**-attribute according to the value that is delivered by *usage expression.*

static conditions: The mode of *access location* must have an **index** mode if an *index expression* is specified; the class of the value delivered by *index expression* must be **compatible** with that **index** mode.

The class of the value delivered by *usage expression* must be **compatible** with the *USAGE*-derived class.

The class of the value delivered by *where expression* must be **compatible** with the *WHERE*-derived class.

dynamic conditions: *CONNECT* causes the *NOTASSOCIATED* exception if *association location* does not contain an association.

CONNECT causes the *CONNECTFAIL* exception if one of the following conditions occurs:

- the association's **existing**-attribute is *FALSE*;
- the association's **readable**-attribute is *FALSE* and *usage expression* delivers *READONLY* or *READWRITE* ;
- the association's **writeable**-attribute is *FALSE* and *usage expression* delivers *WRITEONLY* or *READWRITE* ;
- the association's **indexable**-attribute is *FALSE* and *access location* has an **index** mode;
- the association's **sequencible**-attribute is *FALSE* and *access location* has no **index** mode;
- *where expression* delivers *SAME* , while the association contained in *association location* is not connected to an access location;
- the association's **varying**-attribute is *FALSE* and the *access location* has a dynamic record mode, while *usage expression* delivers *WRITEONLY* or *READWRITE* ;
- the association's **varying**-attribute is *TRUE* and the *access location* has a static record mode, while *usage expression* delivers *READONLY* or *READWRITE* ;
- the *access location* has no **index** mode, while *usage expression* delivers *READWRITE* ;

- the association contained in association *location* cannot be connected to the access *location*, due to implementation defined conditions.

CONNECT causes the *RANGEFAIL* exception if the **index** mode of access *location* is a range mode and the *index expression* delivers a value which lies outside the bounds of that range mode.

example:

20.22 *CONNECT (record_file, file_association, READWRITE);* *(1.1)*
20.24 *READONLY* *(2.1)*

7.4.7 Disconnecting an access location

syntax:

<disconnect io CHILL simple built-in routine call> ::= *(1)*
DISCONNECT (<access location>) *(1.1)*

semantics: *DISCONNECT* disconnects the access location denoted by access *location* from the association it is connected to.

dynamic conditions: *DISCONNECT* causes the *NOTCONNECTED* exception if the access *location* is not connected to an association.

7.4.8 Accessing attributes of access locations

syntax:

<access attr io CHILL value built-in routine call> ::= *(1)*
GETASSOCIATION (<access location>) *(1.1)*
| *GETUSAGE (<access location>)* *(1.2)*
| *OUTOFFILE (<access location>)* *(1.3)*

semantics: *GETASSOCIATION* returns a reference value to the association location that the access *location* is connected to; it returns *NULL* if the access *location* is not connected to an association.

GETUSAGE returns the value of the **usage**-attribute; i.e., *READONLY* (*WRITEONLY*) if the access *location* is connected only for read (write) operations, or *READWRITE* if the access *location* is connected for both read and write operations.

OUTOFFILE returns the value of the **outoffile**-attribute of access *location*; i.e., *TRUE* if the last read operation calculated a transfer index that was not in the file, *FALSE* otherwise.

static properties: The class of a *GETASSOCIATION* built-in routine call is the *ASSOCIATION*-reference class.

The class of an *OUTOFFILE* built-in routine call is the *BOOL* -derived class.

The class of a *GETUSAGE* built-in routine call is the *USAGE*-derived class.

dynamic conditions: *GETUSAGE* and *OUTOFFILE* cause the *NOTCONNECTED* exception if the access *location* is not connected to an association.

example:

21.47 *OUTOFFILE (infiles (FALSE))* *(1.3)*

7.4.9 Data transfer operations

syntax:

<*readrecord io CHILL value built-in routine call*> ::= (1)
READRECORD (<*access location*>[,<*index expression*>] [,<*store location*>]) (1.1)

<*writerecord io CHILL simple built-in routine call*> ::= (2)
WRITERECORD (<*access location*>[,<*index expression*>], <*write expression*>) (2.1)

<*store location*> ::= (3)
<*static mode location*> (3.1)

<*write expression*> ::= (4)
<*expression*> (4.1)

N.B. - The syntax of the *READRECORD* built-in routine is ambiguous but is resolved using the mode of *access location*.

semantics: For the transfer of data to or from a file, the built-in routines *WRITERECORD* and *READRECORD* are defined. The *access location* must have a record mode, and it must be connected to an association in order to transfer data to or from the file that is attached to that association. The transfer direction must not be in contradiction with the actual value of the *access location*'s **usage**-attribute.

Before a transfer takes place, the **transfer** index; i.e., the position in the file of the record to be transferred, is calculated. If the *access location* has no **index** mode, the **transfer** index is the **current** index incremented by 1; if the *access location* has an **index** mode, the **transfer** index is calculated as follows :

transfer index := **base** index + *NUM (v)* − *NUM (l)* + *1*

where *l* is the **lower bound** of the mode of the *access location*'s **index** mode and *v* denotes the value that is delivered by *index expression*. If the transfer of the record with the calculated **transfer** index has been performed successfully, the **current** index becomes the **transfer** index.

The read operation:.

READRECORD transfers data from a file in the outside world to the CHILL program.

If the calculated **transfer** index is not in the file, the **outoffile**-attribute is set *TRUE*; otherwise, the file is positioned and the record is read.

The record that is read must not deliver an **undefined** value; the effect of the read operation is implementation defined if the record being read from the file is not a legal value according to the record mode of the *access location*.

If a *store location* is specified, then the value of the record that was read is assigned to this location. If no *store location* is specified, the value will be assigned to an implicitly created location; the lifetime of this location ends when the *access location* is disconnected or reconnected. Whether the referenced location is created only once by the connect operation, or every time a read operation is performed, is not defined.

READRECORD returns in both cases a reference value that refers to the (possibly dynamic mode) location to which the value was assigned.

The write operation:

WRITERECORD transfers data from the CHILL program to a file in the outside world. The file is positioned to the record with the calculated index and the record is written.

After the record has been written successfully, the number of records is set to the transfer index, if the latter is greater than the actual number of records.

The record written by *WRITERECORD* is the value delivered by *write expression*; the class of this value may be dynamic only if the *access location* has a dynamic record mode.

static properties: The class of the *READRECORD* built-in routine call is the M-reference class, where M is the record mode of the *access location*, if it has a static record mode, or a dynamically parameterised version of it, if the location has a dynamic record mode; the parameters of such a dynamically parameterised record mode are :

- the dynamic **string length** of the string value that was read in case of a string mode;
- the dynamic **upper bound** of the array value that was read in case of an array mode;
- the list of (tag) values associated with the mode of the structure value that was read in case of a **variant** structure.

static conditions: The *access location* must have a record mode.

An *index expression* may not be specified if *access location* has no index mode and must be specified if the *access location* has an index mode; the class of the value delivered by *index expression* must be **compatible** with that index mode.

If *store location* is specified, then the mode of *store location* and the record mode of *access location* must be **equivalent**. The *store location* must be **referable**.

The mode of *store location* must not have the **read-only property**.

The class of the value delivered by *write expression* must be **compatible** with the record mode of the *access location*; the class may be dynamic if and only if the *access location* has a dynamic record mode.

dynamic conditions: The *READRECORD* and *WRITERECORD* built-in routine call cause the *NOTCONNECTED* exception if the *access location* is not connected to an association.

The *READRECORD* or *WRITERECORD* built-in routine call cause the *RANGEFAIL* exception if the index mode of *access location* is a range mode and the *index expression* delivers a value that lies outside the bounds of that range mode.

The *READRECORD* built-in routine call causes the *READFAIL* exception if one of the following conditions occurs :

- the value of the **usage**-attribute is *WRITEONLY* ;
- the value of the **outoffile**-attribute is *TRUE*;
- the reading of the record with the calculated index fails, due to outside world conditions.

The *WRITERECORD* built-in routine call causes the *WRITEFAIL* exception if one of the following conditions occurs :

- the value of the **usage**-attribute is *READONLY* ;
- the writing of the record with the calculated index fails, due to outside world conditions.

The *WRITERECORD* built-in routine call causes the *RANGEFAIL* exception if the access location has a record mode that is a range mode and the *write expression* delivers a value which lies outside the bounds of this range mode.

If the *RANGEFAIL* exception or the *NOTCONNECTED* exception occur then it occurs before the value of any attribute is changed and before the file is positioned.

example:

```
20.24   READRECORD (record_file, curindex, record_buffer);   (1.1)
22.25   READRECORD (fileaccess);                             (1.1)
20.32   WRITERECORD (record_file, curindex, record_buffer);  (2.1)
```

21.61	*WRITERECORD (outfile, buffers(flag));*	*(2.1)*
20.24	*record_buffer*	*(3.1)*
21.61	*buffers(flag)*	*(4.1)*

8 PROGRAM STRUCTURE

8.1 GENERAL

The *do action*, *begin-end block*, *module*, *region*, *spec module*, *spec region*, *quasi module*, *quasi region*, *context*, *receive case action*, *procedure definition* and *process definition* determine the program structure; i.e., they determine the scope of names and the lifetime of locations created in them.

- The word **block** 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* and *parameter spec* of all *formal parameters* of the *formal parameter list*;
  - the *process definition* excluding the *parameter spec* of all *formal parameters* of the *formal parameter list*;
  - the *action statement list* in a *buffer receive alternative* or in a *signal receive alternative*, including the *defining occurrence* of *defining occurrence 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
  - a *module* or *region*, excluding the *contexts*, *defining occurrence* and/or *handler*, if any;
  - a *quasi module* or a *quasi region*;
  - a *spec module* or *spec region*, excluding the *contexts*, if any;
  - a *context*.
- 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 that is not surrounded (see section 8.2) by an inner group of the group (i.e., the part consisting of the outermost nesting level of the group).

A group influences the scope of each *name* **created** in its reach.
Names are created by *defining occurrences* :

- A *defining occurrence* in the *defining occurrence list* of a *declaration*, *mode definition* or *synonym definition* or appearing in a *signal definition* creates a *name* in the reach where the *declaration*, *mode definition*, *synonym definition* or *signal definition*, respectively, is placed.
- A *defining occurrence* in a *set mode* creates a name in the reach directly enclosing the *set mode*.
- A *defining occurrence* appearing in the *defining occurrence list* in a *formal parameter list* creates a name in the reach of the associated *procedure definition* or *process definition*.
- A *defining occurrence* in front of a colon followed by an *action*, *region*, *quasi module*, *quasi region*, *procedure definition*, *entry definition* or *process definition* creates a name in the reach where the *action*, *region*, *quasi module*, *quasi region*, *procedure definition*, *procedure definition* containing the *entry definition*, *process definition*, respectively, is placed.
- A (virtual) *defining occurrence* introduced by a *with part* or in a *loop counter* creates a name in the reach of the block of the associated *do action*.

- A *defining occurrence* in the *defining occurrence list* of a *buffer receive alternative* or a *signal receive alternative* creates a name in the reach of the block of the associated *buffer receive alternative* or *signal receive alternative* respectively.

- A (virtual) *defining occurrence* for a language predefined or an implementation defined name creates a name in the reach of the imaginary outermost process (see section 8.8).

The places where a *name* is used are called **applied occurrences** of the *name*. The name **binding** rules associate a *defining occurrence* with each applied occurrence of the *name* (see section 10.2.2).

A name has a certain scope, i.e. that part of the program where its definition or declarations can be seen and, as a consequence, where it may be freely used. The name is said to be **visible** in that part. Locations and procedures have a certain **lifetime**, i.e. that part of the program where they exist. Blocks determine both visibility of names and the lifetime of the locations created in them. 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.

8.2 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>* | *<spec region>* } *
<action statement list> *(4.1)*

<region body> ::= *(5)*
{ *<data statement>* | *<visibility statement>*} * *(5.1)*

<spec module body> ::= *(6)*
{ *<quasi data statement>* | *<visibility statement>* | *<quasi module>* | *<spec module>*
|
<quasi region> | *<spec region>* | *<quasi cause statement>* } * *(6.1)*

<spec region body> ::= *(7)*
{ *<quasi data statement>* | *<visibility statement>* | *<quasi cause statement>* }* *(7.1)*

<context body> ::= *(8)*
{ *<quasi data statement>* | *<visibility statement>* | *<quasi module>* | *<spec module>*
|
<quasi region> | *<spec region>* } * *(8.1)*

<quasi module body> ::= *(9)*
{ *<quasi data statement>* | *<visibility statement>* | *<quasi module>* | *<spec module>*
|
<quasi region> | *<spec region>* }* *(9.1)*

<quasi region body> ::= *(10)*
{ *<quasi data statement>* | *<visibility statement>* } * *(10.1)*

<action statement list> ::= *(11)*
{ *<action statement>*} * *(11.1)*

<*data statement list*> ::= (12)
{ <*data statement*>} * (12.1)

<*data statement*> ::= (13)
<*declaration statement*> (13.1)
| <*definition statement*> (13.2)

<*definition statement*> ::= (14)

<*synmode definition statement*> (14.1)
| <*newmode definition statement*> (14.2)
| <*synonym definition statement*> (14.3)
| <*procedure definition statement*> (14.4)
| <*process definition statement*> (14.5)
| <*signal definition statement*> (14.6)
| <*empty*>; (14.7)

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 is directly enclosed in zero or more groups as follows:

- If the reach is the reach of a *do action*, *begin-end block*, *procedure definition*, *process definition*, then it is directly enclosed in the group in whose reach the *do action*, *begin-end block*, *procedure definition* or *process definition*, respectively, is placed, and only in that group.
- If the reach is the *action statement list*, or a *buffer receive alternative*, or *signal receive alternative*, or the *action statement list* following **ELSE** in a *receive buffer case action* or *receive signal case action*, then it is directly enclosed in the group in whose reach the *receive buffer case action* or *receive signal case action* is placed, and only in that group.
- If , in a *handler* which is **not** appended to a group, the reach is the *action statement list* in an *on-alternative* or the *action statement list* following **ELSE** , then it is directly enclosed in the group in whose reach the statement to which the *handler* is appended, is placed, and only in that group.
- If the reach is an *on-alternative* or *action statement list* after **ELSE** of a *handler* which is appended to a group, then it is directly enclosed in the group to which the *handler* is appended, and only in that group.
- If the reach is a *module*, *region*, *spec module* or *spec region*, then it is directly enclosed in the group in whose reach it is placed, and also directly enclosed in the context directly in front of the *module*, *region*, *spec module* or *spec region*, if any. This is the only case where a reach has more than one directly enclosing group.
- If the reach is a *context* then it is directly enclosed in the *context* directly in front of it. If there is no such *context*, it has no directly enclosing group.

A reach has **directly enclosing reaches** that are the reaches of the directly enclosing groups. A statement has a unique directly enclosing group, namely, is the group in which the statement is placed. A reach is said to directly enclose a group (reach) if and only if the reach is a directly enclosing reach of the group (reach).

A statement (reach) is said to be surrounded by a group if and only if either the group is the directly enclosing group of the statement (reach) or a directly enclosing reach is surrounded by the group.

A reach is said to be entered when:

- Module reach: the module is executed as an action (e.g., the module is not said to be entered when a goto action transfers control to a label name defined inside the module).
- Begin-end reach: the begin-end block is executed as an action.
- Region reach: the region is encountered (e.g., the region is not said to be entered when one of its **critical** procedures is called).
- Procedure reach: the procedure is entered via 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.

8.3 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 lifetimes of locally created locations (see sections 8.9 and 10.2).

dynamic conditions: A *SPACEFAIL* exception occurs if the *begin-end block* requires local storage for which storage requirements cannot be satisfied.

examples: see 15.73 - 15.90

8.4 PROCEDURE DEFINITIONS

syntax:

<procedure definition statement> ::= (1)
<defining occurrence> : *<procedure definition>*
[*<handler>*] [*<simple name string>*]; (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)
<defining occurrence 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)
<defining occurrence> : *<entry definition>*; (7.1)

<entry definition> ::= (8)
ENTRY (8.1)

derived syntax: A *formal parameter*, where *defining occurrence list* consists of more than one *defining occurrence*, is derived from several *formal parameter* occurrences, separated by commas, one for each *defining occurrence* and each with the same *parameter spec*. For example: *i, j INT* **LOC** is derived from *i INT* **LOC** , *j INT* **LOC** .

semantics: A procedure definition 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 by 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 defining occurrence in front of the procedure definition defines the name of the procedure.

A procedure may have multiple entry points by means of entry statements. These statements are considered to be additional procedure definitions. The defining occurrence 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 defining occurrence><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 pass by location parameter passing, a (possibly dynamic mode) location is passed as a parameter to the procedure body. Only **referable** locations can be passed in this way. The effect is as if at the entry point of the procedure the loc-identity declaration statement: **DCL** <*formal parameter defining occurrence*><*mode*> **LOC** [**DYNAMIC**] := <*actual parameter*>; were encountered. However, such a declaration cannot cause an exception inside the procedure body.

If a *value* is specified that is not a *location*, a location containing the specified value will be implicitly created and passed at the point of the call. The lifetime of the created location is the procedure call. The mode of the created location is dynamic if the value has a dynamic class.

result transmission:

Both a value and a location may be returned from the procedure. In the first case, a *value* is specified in any *result action*, in the latter case, a *location* (see 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 4.2.11) nor as a value procedure call (see section 5.2.12), 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 the specification is 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 *defining occurrence* in a *procedure definition statement* and an *entry statement* defines a **procedure** name.

A **procedure** name has a *procedure definition* attached that 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, as defined by its *procedure definition*:

- It has a list of **parameter specs** that are defined by the *parameter spec* occurrences in the *formal parameter list*, each parameter consisting of a mode, possibly a parameter attribute and/or **register** name.
- It has possibly a **result spec**, consisting of a mode, possibly a result 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** that is, if *generality* is specified, either **general** or **simple** or **inline**, depending on whether **GENERAL** , **SIMPLE** or **INLINE** is specified; otherwise, an implementation default specifies **general** or **simple**. If the **procedure** name is defined inside a 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 9.2.1) the recursivity is **non-recursive**.

A **procedure** name that is **general** is a **general procedure** name. A **general procedure** name has a procedure mode attached, formed as:
PROC *([<parameter list>])* [*<result spec>*]
[**EXCEPTIONS** *(<exception list>)*] [**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 commas.

A name defined in a *defining occurrence list* in the *formal parameter* is a **location** name if and only if the *parameter spec* in the *formal parameter* does not contain the **LOC** attribute. If it does contain the **LOC** attribute, it is a **loc-identity** name. Any such a **location** name or **loc-identity** name is **referable**.

static conditions: If a **procedure** name is **intra-regional** (see section 9.2.2), its procedure definition must not specify **GENERAL** .

If a **procedure** name is **critical** (see section 9.2.1), its definition may specify neither **GENERAL** nor **RECURSIVE** .

No procedure definition may specify both **INLINE** and **RECURSIVE** .

If specified, the *simple name string* must be equal to the name string of the *defining occurrence* in front of the *procedure definition*.

Only if **LOC** is specified in the *parameter spec* or *result spec* may the mode in it have the **non-value property**.

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

examples:

1.4 *add:*
PROC *(i,j INT) (INT)* **EXCEPTIONS** *(OVERFLOW);*
RESULT *i+j;*
END *add;* *(1.1)*

8.5 PROCESS DEFINITIONS

syntax:

<process definition statement> ::= *(1)*
<defining occurrence> : *<process definition>*
[*<handler>*] [*<simple name string>*]; *(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 9).

static properties: A *defining occurrence* in a *process definition statement* defines a **process** name.

static conditions: If specified, the *simple name string* must be equal to the name string of the *defining occurrence* in front of the *process definition*.

A *process definition statement* must not be surrounded by a region or by a block other than the imaginary outermost process definition (see section 8.8).

The parameter attributes in the *formal parameter list* must not be **INOUT** nor **OUT** .

Only if **LOC** is specified in the *parameter spec* in a *formal parameter* in the *formal parameter list* may the mode in it have the **non-value property**.

examples:

14.13

```
PROCESS ();
 wait:
  PROC (x INT );
     /*some wait action*/
  END wait;
  DO FOR EVER ;
     wait(10 /* seconds */ );
      CONTINUE operator_is_ready;
  OD ;
END                                                    (2.1)
```

8.6 MODULES

syntax:

```
<module> ::=                                                          (1)
        [ <contexts> ] [ <defining occurrence>:]
        MODULE <module body> END [ <handler> ] [ <simple name string> ];   (1.1)
      | [ <contexts> ] <remote module>                                (1.2)
```

semantics: A module is an action statement possibly containing local declarations and definitions. A module is a means of restricting the visibility of name strings; it does not influence the lifetime of the locally declared locations.

The detailed visibility rules for modules are given in section 10.2.

static properties: A *defining occurrence* in a *module* defines a **module** name as well as a **label** name. The name has the *module* (seen as a **modulion** i.e., excluding the *contexts*, *defining occurrence*, *handler*, if any) attached.

static conditions: If specified, the *simple name string* must be equal to the name string of the *defining occurrence*.

examples:

7.48

```
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)
```

8.7 REGIONS

syntax:

```
<region> ::=                                                          (1)
        [ <contexts> ] [ <defining occurrence> :] REGION <region body> END
        [ <handler> ] [ <simple name string> ];                       (1.1)
      | [ <contexts> ] <remote region>;                               (1.2)
```

semantics: A region is a means of providing mutually exclusive access to its locally declared data objects for the concurrent executions of processes (see chapter 9). It determines visibility of locally created names in the same way as a module.

static properties: A *defining occurrence* in a *region* defines a **region** name. It has the region (seen as a **modulion** i.e., excluding the *contexts*, *defining occurrence*, *handler*, if any) attached.

static conditions: If specified the *simple name string* must be equal to the name string of the *defining occurrence.*

A *region* must not be surrounded by a block other than the imaginary outermost *process definition.*

examples: see 13.1 - 13.28

8.8 PROGRAM

syntax:

<*program*> ::= *(1)*

{ <*module*> | <*spec module*> | <*region*> | <*spec region*>} $^{+}$ *(1.1)*

semantics: Programs consist of a list of modules or regions surrounded by an imaginary outermost process definition.

The definitions of the CHILL pre-defined names (see Appendix C2) and the implementation defined built-in routines, modes and register names are considered, for lifetime purposes, to be defined in the reach of the imaginary outermost process definition. For their visibility see section 10.2.

8.9 STORAGE ALLOCATION AND LIFETIME

The time during which a location or procedure exists within its program is its **lifetime.**

A location is created by a declaration or by the execution of a *GETSTACK* or an *ALLOCATE* built-in routine call.

The lifetime of a location declared in the reach of a block is the time during which control lies in that block or in a procedure which call originated from that block, unless it is declared with the attribute **STATIC** . The lifetime of a location declared in the reach of a modulion is the same as if it were declared in the reach of the closest surrounding block of the modulion. The lifetime of a location declared with the attribute **STATIC** is the same as if it were declared in the reach of the imaginary outermost process definition. This implies that for a location declaration with the attribute **STATIC** , storage allocation is made only once, namely, when starting the imaginary outermost process. If such a declaration appears inside a procedure definition or process definition, only one location will exist for all invocations or activations.

The lifetime of a location created by executing the *GETSTACK* built-in routine call is the time during which control lies in the directly enclosing block or in a procedure which call originated from that block.

The lifetime of a location created by an *ALLOCATE* built-in routine call is the time starting from the *ALLOCATE* call until the time that the location cannot be accessed anymore by any CHILL program. The latter is always the case if a *TERMINATE* built-in routine call is done on an **allocated** reference value that references the location.

The lifetime of an access created in a loc-identity declaration is the directly enclosing block of the loc-identity declaration.

The lifetime of a procedure is the directly enclosing block of the procedure definition.

static properties: A *location* is said to be **static** if and only if it is a *static mode location* of one of the following kinds:

- A *location name* that is declared with the attribute **STATIC** or whose definition is not surrounded by a block other than the imaginary outermost process definition.
- A *string element* or *string slice* where the *string location* is **static** and either the *left element* and *right element*, or *start element* and *slice size* are **constant**.

- An *array element* or *array slice* where the *array location* is **static** and either the *lower element* and the *upper element*, or the *first element* and *slice size* 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**.

8.10 CONSTRUCTS FOR PIECEWISE PROGRAMMING

8.10.1 Remote pieces

syntax:

<remote module> ::= *(1)*
[*<simple name string>*:] **MODULE REMOTE** *<source text designator>;* *(1.1)*

<remote region> ::= *(2)*
[*<simple name string>*:] **REGION REMOTE** *<source text designator>;* *(2.1)*

<remote spec module> ::= *(3)*
[*<simple name string>*:] **SPEC MODULE REMOTE** *<source text designator>;* *(3.1)*

<remote spec region> ::= *(4)*
[*<simple name string>*:] **SPEC REGION REMOTE** *<source text designator>;* *(4.1)*

<remote context> ::= *(5)*
CONTEXT REMOTE *<source text designator>* **FOR** *(5.1)*

<source text designator> ::= *(6)*
<character string literal> *(6.1)*
| *<text reference name>* *(6.2)*
| *<empty>* *(6.3)*

semantics: *Remote modules*, *remote regions*, *remote spec modules*, *remote spec regions* and *remote contexts* are means to represent the source text of a program as a set of (interconnected) files.

A *source text designator* refers in an implementation defined way to a piece of CHILL source text, as follows :

- If the *source text designator* is empty, the source text is retrieved from the structure of the program in which it is placed.

- If the *source text designator* contains a *character string literal*, the *character string literal* is used to retrieve the text.

- If the *source text designator* contains a *text reference name*, the *text reference name* is interpreted in an implementation defined way to retrieve the source text.

A program with *remote modules* (*remote regions*, *remote spec modules*, *remote spec regions*, *remote contexts*) is equivalent to the program built by replacing each *remote module* (*remote region*, *remote spec module*, *remote spec region*, *remote context*) by the piece of CHILL text referred to by its source text designator.

static conditions: The *source text designator* in a (1. *remote module*, 2. *remote region*, 3. *remote spec module*, 4. *remote spec region*, 5. *remote context*) must refer to a piece of source text which is a terminal production of a (1. a *module* which is not a *remote module*, 2. *region* which is not a *remote region*, 3. *spec module* which is not a *remote spec module*, 4. *spec region* which is not a *remote spec region*, 5. *context* which is not a *remote context*).

When the source text referred to by the *source text designator* in a (1. *remote module*, 2. *remote region*) starts with a *defining occurrence*, then the (1. *remote module*, 2. *remote region*) must start with a *simple name string* which is the name string of that *defining occurrence*.

When the source text referred to by the *source text designator* in a (1. *remote spec module*, 2. *remote spec region*) starts with a *simple name string*, then the (1. *remote spec module*, 2. *remote spec region*) must start with the same *simple name string*.

examples:

25.9 **MODULE REMOTE** *stack_code* *(1.1)*
25.9 *stack_code* *(6.2)*

8.10.2 Spec modules, spec regions and contexts

syntax:

<spec module> ::= *(1)*
[*<contexts>*] [*<simple name string>* :] **SPEC MODULE**
<spec module body> **END** [*<simple name string>*]; *(1.1)*
| *<remote spec module>* *(1.2)*

<spec region> ::= *(2)*
[*<contexts>*] [*<simple name string>* :] **SPEC REGION**
<spec region body> **END** [*<simple name string>*]; *(2.1)*
| *<remote spec region>* *(2.2)*

<contexts> ::= *(3)*
<context> { *<context>* } * *(3.1)*

<context> ::= *(4)*
CONTEXT *<context body>* **END** [*<quasi handler>*] **FOR** *(4.1)*
| *<remote context>* *(4.2)*

semantics: Spec modules, spec regions and contexts are used to specify static properties of names. They are redundant but they can be used for piecewise programming.

Simple name strings in spec modules and spec regions are not names, they are not bound, and they have no visibility rules.

static conditions: In a *spec module* or a *spec region*, the optional *simple name string* following **END** may only be present if the optional *simple name string* before **SPEC** is present. When both are present, they must have equal name strings.
A *context* which has no directly enclosing group may not contain visibility statements.

examples:

23.1 *letter_count:*
SPEC MODULE
SEIZE *max* ;
count: **PROC** (**ROW** *CHAR (max)* **IN** , **ARRAY** *('A':'Z') INT* **OUT**); **END** ;
GRANT *count* ;
END *letter_count;* *(1.1)*

24.1 **CONTEXT**
count: **PROC** (**ROW** *CHAR (max)* **IN** , **ARRAY** *('A':'Z') INT* **OUT**); **END** ;
END FOR *(4.1)*

8.10.3 Quasi statements

syntax:

<quasi data statement> ::= *(1)*
<quasi declaration statement> *(1.1)*
| *<quasi definition statement>* *(1.2)*

<*quasi declaration statement*> ::= (2)
DCL <*quasi declaration*> {, <*quasi declaration*> } *; (2.1)

<*quasi declaration*> ::= (3)
<*defining occurrence list*> <*mode*>
[**STATIC**] [**NONREF**] [**DYNAMIC**] (3.1)

<*quasi definition statement*> ::= (4)
<*synmode definition statement*> (4.1)
| <*newmode definition statement*> (4.2)
| <*synonym definition statement*> (4.3)
| <*quasi procedure definition statement*> (4.4)
| <*quasi process definition statement*> (4.5)
| <*signal definition statement*> (4.6)
| <*empty*>; (4.7)

<*quasi procedure definition statement*> ::= (5)
<*defining occurrence*> : **PROC** ([<*quasi formal parameter list*>])
[<*result spec*>] [**EXCEPTIONS** (<*exception list*>)]
<*procedure attributes*> { <*quasi entry statement*> } *
END [<*simple name string*>]; (5.1)

<*quasi entry statement*> ::= (6)
<*defining occurrence*> : **ENTRY** ; (6.1)

<*quasi formal parameter list*> ::= (7)
<*quasi formal parameter*> {, <*quasi formal parameter*> } * (7.1)

<*quasi formal parameter*> ::= (8)
[<*simple name string*> {, <*simple name string*> } *] <*parameter spec*> (8.1)

<*quasi process definition statement*> ::= (9)
<*defining occurrence*> : **PROCESS** ([<*quasi formal parameter list*>])
END [<*simple name string*>]; (9.1)

<*quasi region*> ::= (10)
[<*defining occurrence*> :] **REGION** <*quasi region body*>
END [<*simple name string*>]; (10.1)

<*quasi module*> ::= (11)
[<*defining occurrence*> :] **MODULE** <*quasi module body*>
END [<*simple name string*>]; (11.1)

<*quasi cause statement*> ::= (12)
CAUSE <*exception list*>; (12.1)

<*quasi handler*> ::= (13)
ON ELSE END
| **ON** <*exception list*> [**ELSE**] **END** (13.1)

semantics: Quasi statements are used in spec modules, spec regions and contexts to specify static properties of names. These specifications are redundant, but quasi statements can be used for piecewise programming.

Quasi cause statements indicate the presence of cause statements in remote modules or remote regions directly enclosed in the reach directly enclosing the reach of the spec module or spec region in which the quasi cause statement is placed.

Quasi handlers indicate the presence of a handler in the program, reachable from the module region, or context directly enclosed in the context to which the quasi handler is appended.

static properties: *Quasi statements* are restricted forms of the corresponding *statements*, and have the same static properties.

The name defined by a *defining occurrence* in a *quasi declaration* is **referable** if **NONREF** is not specified.

A *defining occurrence* in front of **REGION** in a *quasi region* (in front of **MODULE** in a *quasi module*) defines a **quasi region** (**quasi module**) name with the *quasi region* (*quasi module*) attached.

All *defining occurrences* surrounded by a *context*, a *spec module* or a *spec region* are **quasi** *defining occurrences*.

static conditions: Quasi statements are restricted forms of the corresponding statements and are subject to their static conditions.

8.10.4 Matching between quasi defining occurrences and defining occurrences

The following rules apply :

- If a *name string* in a reach that is not the reach of a *quasi module*, *quasi region*, *spec module*, *spec region* or *context* is bound to a **quasi** *defining occurrence*, then it must also be bound to a *defining occurrence* which is not a **quasi** *defining occurrence*. The two *defining occurrences* must have identical static properties (semantic category, referability, regionality, staticity, forbidden fields, ..., etc, when applicable). If the two *defining occurrences* have a mode attached, or if their static properties involve a mode, then those two modes must be **alike**. If the **quasi** *defining occurrence* is enclosed in the reach then the *defining occurrence* which is not a **quasi** *defining occurrence* must be surrounded by, but not directly enclosed in, the reach.

- In every reach, if *name strings* N1 and N2 are bound respectively to *defining occurrences* D1 and D2 and **quasi** *defining occurrences* Q1 and Q2 and if the modes of Q1 and Q2 are **N-alike** then the modes of D1 and D2 must also be **N-alike**.

9 CONCURRENT EXECUTION

9.1 PROCESSES AND THEIR DEFINITIONS

A **process** is the sequential execution of a series of statements. It may be executed concurrently with other processes. The behaviour of a process is described by a **process definition** (see section 8.5), that describes the objects local to a process and the series of action statements to be executed sequentially.

A process is **created** by the evaluation of a start expression (see section 5.2.14). 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 8.9), 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 8.8), 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.

9.2 MUTUAL EXCLUSION AND REGIONS

9.2.1 General

Regions (see section 8.7) are a means of providing processes with mutually exclusive access to locations declared in them. Static context conditions (see section 9.2.2) are made such that accesses by a process (which is not the imaginary outermost process) to locations declared in a region can be made only by calling procedures that 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 either it is defined inside a region and granted by the region or if a **procedure** name with the same procedure definition (see section 8.4) 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 or 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 that causes the executing process to become delayed (see section 9.3). In the case of dynamically nested critical procedure calls, only the latest locked region will be released.

- The process executing the **critical** procedure terminates. In the case of dynamically nested **critical** procedure calls, all the regions locked by the process will be released.

If, while the region is locked, a process attempts to call one of its **critical** procedures or is re-activated, the process is suspended until the region is released. (Note that the process remains active in the CHILL sense).

When a region is released and more than one process has been suspended while attempting 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 lock the region according to an implementation defined scheduling algorithm.

9.2.2 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);

- **intra-regional** procedures are not **general** (see section 8.4);

- **critical** procedures are neither **general** nor **recursive** (see section 8.4).

A *location* and *procedure call* have a **regionality** which is **intra-regional** or **extra-regional**. A *value* has a **regionality** which is **intra-regional** or **extra-regional** or nil. These properties are defined as follows:

1. **Location**

   A *location* is **intra-regional** if and only if any of the following conditions is fulfilled:

   - It is an *access name* that is either:

     - a *location name* declared textually inside a *region* and not defined in a *formal parameter* of a **critical** procedure,

     - a *loc-identity name*, where the *location* in its declaration is **intra-regional** or that is defined in a *formal parameter* of an **intra-regional** procedure,

     - a *based name* where the *bound or free reference location name* in its declaration is **intra-regional**,

     - a *location enumeration* name, where the *array location* or *string location* in the associated *do action* is **intra-regional**,

     - a *location do-with name*, where the *structure location* in the associated *do action* is **intra-regional**.

   - It is a *dereferenced bound reference*, where the *bound reference primitive value* in it is **intra-regional**.

   - It is a *dereferenced free reference*, where the *free reference primitive value* in it is **intra-regional**.

   - It is a *dereferenced row*, where the *row primitive value* in it is **intra-regional**.

   - It is an *array element* or *array slice*, where the *array location* in it is **intra-regional**.

   - It is a *string element* or *string slice*, where the *string location* in it is **intra-regional**.

   - It is a *structure field*, where the *structure location* in it is **intra-regional**.

- It is a *location procedure call*, where in the *location procedure call* a *procedure name* is specified which is **intra-regional.**
- It is a *location built-in routine call*, that the implementation specifies it is **intra-regional.**
- It is a *location conversion*, where the *static mode location* in it is **intra-regional.**

A *location* which is not **intra-regional** is **extra-regional.**

2. **Value**

A *value* has a **regionality** depending on its class. If it has the M-derived class or the **all**-class or the **null**-class then it has **regionality** nil. Otherwise it has the M-value class or the M reference class and it has a **regionality** depending on the mode M as follows :

If M does not have the **referencing property** then the **regionality** is nil; otherwise, the *value* is an *operand-6* (and has the **referencing property**) :

If it is a *primitive value* then

- If it is a *location contents* that is a *location*, then it is that of the *location.*
- It it is a *value name* then
  - if it is a *synonym name* then it is that of the *constant value* in its definition;
  - if it is a *value do-with name* then it is that of the *structure primitive value* in the associated do action;
  - if it is a *value receive name* then it is **extra-regional.**
- If it is a *tuple* then if one of the *value* occurrences in it has **regionality** not nil then it is that of that *value* (it does not matter which choice is made, see section 5.2.5 static conditions); otherwise, it is nil.
- If it is a *value array element* or a *value array slice* then it is that of the *array primitive value* in it.
- If it is a *value structure field* then it is that of the *structure primitive value* in it.
- If it is an *expression conversion* then it is that of the *expression* in it.
- If it is a *value procedure call* then it is that of the *procedure call* in it.
- If it is an *implementation value built-in routine call* then it is defined by the implementation.
- If it is a *CHILL value built-in routine call* then if it is surrounded by a region then it is **intra-regional**; otherwise, it is **extra-regional.**

If it is a *referenced location* then it is that of the *location* in it.

If it is a *receive expression* then it is **extra-regional.**

3. **Procedure name**

A *procedure name* is **intra-regional** if and only if it is defined inside a *region* and it is not **critical** (i.e., not granted by the region). Otherwise it is **extra-regional.**

4. **Procedure call**

A *procedure call* is **intra-regional** if it contains a *procedure name* which is **intra-regional**; otherwise, it is **extra-regional.**

A *value* is **regionally safe** for a non-terminal (used only for *location*, *procedure call* and *procedure name*) if and only if :

- the non-terminal is **extra-regional** and the *value* is not **intra-regional**;
- the non-terminal is **intra-regional** and the *value* is not **extra-regional**.

9.3 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 the 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 the set of delayed processes that is attached to each event location specified in a delay alternative of the delay case action.

Receive expression (see section 5.3.8). 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 the set of delayed receiving processes attached to the specified (empty) buffer location.

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 that 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 the set of delayed processes attached to each signal name specified in the signal-receive alternative.

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 no value is present in any of the specified buffer locations, no sending process is delayed on any of the specified buffer locations, and if **ELSE** is not specified. It becomes a member of the set of delayed receiving processes that is attached to each buffer location specified in a buffer-receive alternative of the receive buffer case action.

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 that 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 it was delayed in the region. The region will then be locked again.

9.4 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 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.

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 9.3).

Receive expression (see section 5.3.8). 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 and from 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, a 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.

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 and from 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 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, a 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 attached to the buffer location and its value is received.

When a process executes an action that 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.

9.5 SIGNAL DEFINITION STATEMENTS

syntax:

<signal definition statement> ::= *(1)*
SIGNAL *<signal definition>* { ,*<signal definition>*} *; *(1.1)*

<signal definition> ::= *(2)*
<defining occurrence> [= (*<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 *defining occurrence* in a *signal definition* defines a a **signal** name.

A **signal** name has the following properties:

- It has an optional list of modes attached, that are the modes mentioned in the *signal definition*.
- It has an optional **process** name attached that is the *process name* specified after **TO** .

static conditions: No *mode* in a *signal definition* may have the **non-value property**.

examples:

15.27 **SIGNAL** *initiate = (INSTANCE),*
terminate; *(1.1)*

10 GENERAL SEMANTIC PROPERTIES

10.1 MODE CHECKING

10.1.1 Properties of modes and classes

10.1.1.1 Read-only property

Informal

A mode has the **read-only property** if it is a **read-only** mode or contains a component or a sub-component etc. which is a **read-only** mode.

Definition

A mode has the **read-only property** if and only if it is :

- an array mode with an **element** mode that has the **read-only property**;
- a structure mode where at least one of its **field** modes has the **read-only property**, where the field is not a **tag** field with an implicit **read-only** mode of a **parameterised** structure mode;
- a **read-only** mode.

10.1.1.2 Parameterisable modes

Informal

A mode is **parameterisable** if it can be parameterised.

Definition

A mode is **parameterisable** if and only if it is

- a **parameterisable variant** structure mode;
- an array mode;
- a string mode.

10.1.1.3 Referencing property

Informal

A mode has the **referencing property** if it is a reference mode or contains a component or a sub-component, etc. that is a reference mode.

Definition

A mode has the **referencing property** if and only if it is :

- an array mode with an **element** mode that has the **referencing property**;
- a structure mode where at least one of its **field** modes has the **referencing property**;
- a reference mode.

10.1.1.4 Tagged parameterised property

Informal

A mode has the **tagged parameterised property** if it is a **tagged parameterised** structure mode or contains a component or a sub-component etc. which is a **tagged parameterised** structure mode.

Definition

A mode has the **tagged parameterised property** if and only if it is :

- an array mode with an **element** mode which has the **tagged parameterised property**;
- a structure mode where at least one of its **field** modes has the **tagged parameterised property**;
- a **tagged parameterised** structure mode.

10.1.1.5 Non-value property

Informal

A mode has the **non-value property** if no expression or primitive value denotation exists for the mode.

Definition

A mode has the **non-value property** if and only if it is :

- an array mode with an **element** mode that has the **non-value property**;
- a structure mode where at least one of its **field** modes has the **non-value property**;
- an event mode, a buffer mode, an access mode or an association mode.

10.1.1.6 Root mode

Any M-value class or M-derived class, where M is a discrete mode or a string mode, has a **root** mode defined as:

- M, if M is not a range mode;
- the **parent** mode of M, if M is a range mode.

10.1.1.7 Resulting class

Given two **compatible** classes (see section 10.1.2.7), 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-value class and the N-value class, the N-derived class or the **all** 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-derived class and the N-derived class or the **all** class is the M-derived class;
- the **resulting class** of the **all** class and the **all** class is the **all** 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 the **resulting class** of the **resulting class** of the list C_i (i=1,...,n–1) and the class C_n if $n > 1$; otherwise, as the **resulting class** of C_1 and C_1 .

(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**.)

10.1.2 Relations on modes and classes

10.1.2.1 General

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 other relations which are mainly used in this chapter for the above mentioned purpose.

10.1.2.2 Equivalence relations on modes

Informal

The following equivalence relations play a role in the formulation of the compatibility relations:

- Two modes are **similar** if they are of the same kind; i.e., they have the same hereditary properties.
- Two modes are **v-equivalent** (value-equivalent) if they are **similar** and also have the same **novelty**.
- Two modes are **equivalent** if they are **v-equivalent** and also possible differences in value representation in storage or minimum storage size are taken into account.
- Two modes are **l-equivalent** (location-equivalent) if they are **equivalent** and also have the same **read-only** specification.
- Two modes are **alike** if they are indistinguishable; i.e., if all operations that can be applied to objects of one of the modes can be applied to the other one as well, provided that **novelty** is not taken into account.
- Two modes are **N-alike** if they are **alike** and have equal **novelty** 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 :

- 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** and for each **set element** name defined by one mode there is a **set element** name defined by the other mode which has the same name string and the same representation value;
- they are range modes with **similar parent** modes;
- one is a range mode whose **parent** mode is **similar** to the other mode;
- one is a boolean mode and the other a **bit** string mode of length 1;

- 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, the same parameter attributes and the same **register** names, if present;
    2. they both have or both do not have a **result spec**. If present, the **result specs** must have **l-equivalent** modes, the same attributes and the same **register** names, 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 **event length** or the same **event length**;
- they are buffer modes such that:
    1. they both have no **buffer length** or the same **buffer length**;
    2. they have **l-equivalent buffer element** modes;
- they are association modes;
- they are access modes such that:
    1. they both have no **index** mode or both have **index** modes which are **equivalent**;
    2. at least one has no **record** mode, or both have **record** modes that are **l-equivalent** and that are both **static record** modes or both **dynamic record** modes.
- they are string modes such that:
    1. they both are **bit** string modes or **character** string modes;
    2. they have the same **string length**. This check is dynamic in the case that one or both modes is (are) dynamic. Check failure will result in the *RANGEFAIL* exception;
- they are array modes such that:
    1. their **index** modes are **v-equivalent**;
    2. their **element** modes are **equivalent**;
    3. their **element layouts** are **equivalent**;
    4. they have the same **number of elements**. This check is dynamic if one or both modes is (are) dynamic. Check failure will result in the *RANGEFAIL* exception;
- they are structure modes which are not **parameterised** structure modes such that:
    1. they have the same number of fields and corresponding (by position) fields are **equivalent**;

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 are 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 one is a boolean mode, the other must also be a boolean mode;
- if one is a character mode, the other must also be a character mode;
- if one is a range mode, the other must also be a range mode and both **upper bound**s must be equal and both **lower bound**s must be equal.

The relation l-equivalent

Two modes are **l-equivalent** if and only if they are **equivalent** and if one is a **read-only** mode, the other must also be a **read-only** mode, and:

- if they are bound reference modes, their **referenced** modes must be **l-equivalent**;
- if they are row modes, their **referenced origin** modes must be **l-equivalent**;
- if they are array modes, their **element** modes must be **l-equivalent**;
- if they are structure modes which are not **parameterised** structure modes, corresponding (by position) fields must be **l-equivalent**; if they are **parameterised** structure modes, their **origin variant** structure modes 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:

- Fixed fields and variant fields

    1. Both fields must have **equivalent field layout**.

    2. Both **field** modes must be 1. **equivalent**, 2. **l-equivalent**.

- Alternative fields

    1. 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.

    2. Both must have the same number of variant alternatives and corresponding (by position) variant alternatives must be 1. **equivalent**, 2. **l-equivalent**.

    3. Both must have no **ELSE** specified or both must have **ELSE** specified. In the latter case, the same number of variant fields must follow and corresponding (by position) variant fields must be 1. **equivalent**, 2. **l-equivalent**.

- Variant alternatives

    1. Both variant alternatives must have the same number of case label lists and corresponding (by position) case label lists must either be both *irrelevant*, or both define the same set of values. If one case label list contains an **ELSE** then so must the other.

    2. Both variant alternatives must have the same number of variant fields and corresponding (by position) variant fields must be 1. **equivalent**, 2. **l-equivalent**.

The relation equivalent for layout

In the rest of the section, it will be assumed that each *pos* is of the form:
POS *(<number>,<start bit>,<length>)*

and that each *step* is of the form:
STEP *(<pos>,<step size>)*

Section 3.11.6 gives the appropriate rules to bring *pos* or *step* in the required form.

- Field layout

  Two **field layouts** are **equivalent** if they are both **NOPACK** , or both **PACK** , or both *pos*. In the latter case the one *pos* must be **equivalent** to the other one (see below).

- Element layout

  Two **element layouts** are **equivalent** if they are both **NOPACK** , both **PACK** , or both *step*. In the latter case the *pos* in the one *step* must be **equivalent** to the *pos* in the other one (see below) and *step size* must deliver the same values for the two **element layouts**.

- Pos

  A *pos* is **equivalent** to another *pos* if and only if both *word* occurrences deliver the same value, both *start bit* occurrences deliver the same value and both *length* occurrences deliver the same value.

The relation alike

Two modes are **alike** if and only if they both are or both are not **read-only** modes and they both have **novelty nil** or both have **novelty** non-**nil** and

- they are integer modes;
- they are boolean modes;
- they are character modes;
- they are **similar** set modes;
- they are range modes with equal **upper bounds** and equal **lower bounds**;
- they are powerset modes such that their **member** modes are **alike**;
- they are bound reference modes such that their **referenced** modes are **alike**;
- they are free reference modes;
- they are row modes such that their **referenced origin** modes are **alike**;
- they are procedure modes such that :

    1. they have the same number of **parameter specs** and corresponding (by position) **parameter specs** have **alike** modes, the same parameter attributes and the same **register** name, if present;

    2. they both have or both do not have a **result spec**. If present, the **result specs** must have **alike** modes, the same attributes and the same **register** name, 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 **event length** or the same **event length**;
- they are buffer modes such that :
    1. they both have no **buffer length** or the same **buffer length**;
    2. they have **buffer element** modes which are **alike**;
- they are association modes;
- they are access modes such that :
    1. they both have no **index** mode or both have **index** modes that are **alike**;
    2. at least one has no **record** mode or both have **record** modes that are **alike** and that are both **static record** modes or both **dynamic record** modes;
- they are string modes such that :
    1. they both are **bit** string modes or **character** string modes;
    2. they have the same **string length**;
- they are array modes such that :
    1. their **index** modes are **alike**;
    2. their **element** modes are **alike**;
    3. their **element layout**s are **equivalent**;
    4. they have the same **number of elements**;
- they are structure modes that are not **parameterised** structure modes such that :
    1. they have the same number of fields and corresponding (by position) fields are **alike**;
    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 **alike**;
    2. their corresponding (by position) values are the same.

The relation alike for fields

Two fields (both fields in the context of two given structure modes) are **alike** if and only if both fields are fixed fields which are **alike** or both are alternative fields which are **alike**.

The relation **alike** is recursively defined for (corresponding) fixed fields, variant fields, alternative fields and variant alternatives respectively in the following way :

- Fixed fields and variant fields
    1. Both fields must have **equivalent field layout**.
    2. Both **field** modes must be **alike**.
    3. Both fields must have the same name string attached.

- Alternative fields

    1. 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.

    2. Both must have the same number of variant alternatives and corresponding (by position) variant alternatives must be **alike**.

    3. Both must have no **ELSE** specified or both must have **ELSE** specified. In the latter case, the same number of variant fields must follow and corresponding (by position) variant fields must be **alike**.

- Variant alternatives

    1. Both variant alternatives must have the same number of case label lists and corresponding (by position) case label lists must either be both *irrelevant*, or both define the same set of values. If one case label list contains an **ELSE** , then so must the other.

    2. Both variant alternatives must have the same number of variant fields and corresponding (by position) variant fields must be **alike**.

The relation "N-alike"

Two modes that are **alike**, are **N-alike** if and only if they have the same **novelty** and if that **novelty** is **nil**, then further :

- if they are powerset modes, their **member** modes must be **N-alike**;
- if they are array modes, their respective **index** modes and **element** modes must be **N-alike**;
- if they are structure modes, their corresponding (by position) **field** modes must be **N-alike**;
- if they are reference modes, their **referenced** modes must be **N-alike**;
- if they are row modes, their **referenced origin** modes must be **N-alike**;
- if they are procedure modes, their **result spec**, if present, and corresponding (by position) **parameter specs**, if any, must have **N-alike** modes;
- if they are buffer modes, their **buffer element** modes must be **N-alike**;
- if they are access modes, their **index** modes, if present, must be **N-alike** and their **record** modes, if present, must be **N-alike**.

10.1.2.3 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 equal or 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 which are not **parameterised** structure modes, any **field** mode of M must be **read-compatible** with the corresponding **field** mode of N. If M and N are **parameterised** structure modes, the **origin variant** structure mode of M must be **read-compatible** with the **origin variant** structure mode of N.

10.1.2.4 The relation dynamic read-compatible

Informal

The relation **dynamic read-compatible** is relevant only for modes that can be dynamic, i.e., string, array and **variant** structure modes. **A parameterisable** mode M is said to be **dynamic read-compatible** with a (possibly dynamic) mode N, if there exists a dynamically parameterised version of M which is **read-compatible** with N.

Definition

A mode M is **dynamic read-compatible** with a mode N (a non-symmetric relation) if and only if one of the following holds :

- M and N are string modes and there exists a length p such that M(p) is **read-compatible** with N. This check is dynamic if N is dynamic. Check failure will result in the *RANGEFAIL* exception.
- M and N are array modes and there exists a value p such that M(p) is **read-compatible** with N. This check is dynamic if N is dynamic. Check failure will result in the *RANGEFAIL* exception.
- M is a **parameterisable variant** structure mode and N is a **parameterised** structure mode and there exists a list of values p_1 , ..., p_n such that M(p_1 , ..., p_n) is **read-compatible** with N. This check is dynamic if N is dynamic. Check failure will result in the *TAGFAIL* exception.
- M and N are **parameterisable variant** structure modes and M is **read-compatible** with N.

10.1.2.5 The relation restrictable

Informal

The relation **restrictable** is relevant for **equivalent** modes with the **referencing property**. A mode M is said to be **restrictable** to a mode N if it or its possible sub-components refer to locations with equal or less restrictive **read-only** specification than those referenced by N. This relation is therefore non-symmetric. The relation is used in assignments.

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 M and N are **equivalent** and further :

- if M and N are bound reference modes, the **referenced** mode of N must be **read-compatible** with the **referenced** mode of M;
- if M and N are row modes, the **referenced origin** mode of N must be **read-compatible** with the **referenced origin** mode of M;
- if M and N are array modes, the **element** mode of M must be **restrictable** to the **element** mode of N;
- if M and N are structure modes, each **field** mode of M must be **restrictable** to the corresponding **field** mode of N.

10.1.2.6 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. The value p must not be greater than the **string length** of V. This check is dynamic if N is a dynamic mode. Check failure will result in a *RANGEFAIL* exception;
     - 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. The value p must not be greater than the **upper bound** of V. This check is dynamic if N is a dynamic mode. Check failure will result in a *RANGEFAIL* exception;
     - 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.

10.1.2.7 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 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. The value p must not be greater than the **string length** of V. This check is dynamic if M is a dynamic mode. Check failure will result in a *RANGEFAIL* exception;

- 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) **upper bound** of M. The value p must not be greater than the **upper bound** of V. This check is dynamic if M is a dynamic mode. Check failure will result in a *RANGEFAIL* exception;

- 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 M.

- The M-derived class is **compatible** with the N-derived class or N-value class if and only if M and N are **similar**.

- The M-value class is **compatible** with the N-value class if and only if M and N are **v-equivalent**.

Two lists of classes are **compatible** if and only if both lists have the same number of classes and corresponding (by position) classes are **compatible**.

10.1.3 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)*
| *<irrelevant>* *(2.2)*

<case label> ::= *(3)*
<discrete literal expression> *(3.1)*
| *<literal range>* *(3.2)*
| *<discrete mode name>* *(3.3)*
| **ELSE** *(3.4)*

<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 3.11.4), 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 or implicitly indicated in the case of **ELSE** ;

- the case label list consists of *irrelevant*.

The values **explicitly** indicated by a case label are the values delivered by any *discrete 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*; if it is **ELSE** , the **all** class.

- A *case label list* has a class attached, which is, if it is *irrelevant*, then the **all** class, otherwise the **resulting class** of the classes of each *case label*.

- A *case label specification* has a list of classes attached, which are the classes of the case label lists.

- A list of case label specifications has a **resulting list of classes** attached. This **resulting list of classes** is formed by constructing, for each position in the list, the **resulting class** of all the classes that have that position.

A list of case label specifications is **complete** if and only if for all lists of possible selector values, a case label specification is present, which matches the list of selector values. The set of all possible selector values is determined by the context as follows:

- For a **tagged variant** structure mode it is the set of values defined by the mode of the corresponding **tag** field.

- For a **tag-less variant** structure mode it is the set of values defined by the **root** mode of the corresponding **resulting class** (this class is never the **all** class, see section 3.11.4).

- 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:

11.9	*(occupied)*	*(3.1)*
11.58	*(rook),(*)*	*(1.1)*
8.25	*(**ELSE**)*	*(2.2)*

10.1.4 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.

10.1.4.1 Names

Mode names

access mode name:	a *name* defined to be an access mode.
array mode name:	a *name* defined to be an array mode.
association mode name:	a *name* defined to be an association mode.
boolean mode name:	a *name* defined to be a boolean mode.
bound reference mode name:	a name defined to be a bound reference mode.
buffer mode name:	a *name* defined to be a buffer mode
character mode name:	a *name* defined to be a character mode.
discrete mode name:	a *name* defined to be a discrete mode.
event mode name:	a *name* defined to be an event mode.
free reference mode name:	a *name* defined to be a free reference mode.
instance mode name:	a *name* defined to be an instance mode.
integer mode name:	a *name* defined to be an integer mode.
mode name:	see section 3.2.1
newmode name:	see section 3.2.3
parameterised array mode name:	a *name* defined to be a **parameterised** array mode.
parameterised string mode name:	a *name* defined to be a **parameterised** string mode.
parameterised structure mode name:	a *name* defined to be a **parameterised** structure mode.
powerset mode name:	a *name* defined to be a powerset mode.
procedure mode name:	a *name* defined to be a procedure mode.
range mode name:	a *name* defined to be a range mode.
row mode name:	a *name* defined to be a row mode.
set mode name:	a *name* defined to be a set mode.
string mode name:	a *name* defined to be a string mode.
structure mode name:	a *name* defined to be a structure mode.
synmode name:	see section 3.2.2
variant structure mode name:	a *name* defined to be a **variant** structure mode.

Access names

based name:	see section 4.1.4
location name:	see sections 4.1.2
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

Value names

synonym name:	see section 5.1
value do-with name:	see section 6.5.4
value enumeration name:	see section 6.5.2
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.
general procedure name:	a *procedure name* whose generality is **general**.
label name:	see section 6.1, 8.6
module name:	see sections 8.6
non-reserved name:	a *name* which is none of the

reserved names mentioned in Appendix C1,

procedure name: see section 8.4
process name: see section 8.5
region name: see sections 8.7
reserved simple name string list: a *simple name string list* consisting solely of **reserved** simple name strings. (see Appendix C1)
set element name: see section 3.4.5
signal name: see section 9.5
tag field name: see section 3.11.4
undefined synonym name: see section 5.1

10.1.4.2 Locations

access location: a *location* with an access mode.
array location: a *location* with an array mode.
association location: a *location* with an association mode.
buffer location: a *location* with a buffer mode.
discrete location: a *location* with a discrete mode.
event location: a *location* with an event mode.
instance location: a *location* with an instance mode.
static mode location: a *location* with a static mode.
string location: a *location* with a string mode.
structure location: a *location* with a structure mode.

10.1.4.3 Expressions and values

array primitive value: a *primitive value* whose class is **compatible** with an array mode.
boolean expression: an *expression* whose class is **compatible** with a boolean mode.
bound reference primitive value: a *primitive value* whose class is **compatible** with a bound reference mode.
constant value: a *value* which is **constant**.
discrete expression: an *expression* whose class is **compatible** with a discrete mode.
discrete literal expression: a *discrete expression* which is **literal**.
free reference primitive value: a *primitive value* whose class is **compatible** with a free reference mode.
instance primitive value: a *primitive value* whose class is **compatible** with an instance mode.
integer expression: an *expression* whose class is **compatible** with an integer mode.
integer literal expression: an *integer expression* which is **literal**.
powerset expression: an *expression* whose class is **compatible** with a powerset mode.
procedure primitive value: a *primitive value* whose class is **compatible** with a procedure mode.
reference primitive value: a *primitive value* whose class is **compatible** with either a bound reference mode, a free reference mode or a row mode.
row primitive value: a *primitive value* whose class is **compatible** with a row mode.
string primitive value: a *primitive value* whose class is **compatible** with a string mode.
structure primitive value: a *primitive value* whose class is **compatible** with a structure mode.

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.

10.1.4.4 Built-in routine calls

access attr io CHILL value built-in routine call	see section 7.4.8
associate io CHILL location built-in routine call	see section 7.4.2
association attr io CHILL value built-in routine call	see section 7.4.4
connect io CHILL simple built-in routine call	see section 7.4.6
disconnect io CHILL simple built-in routine call	see section 7.4.7
dissociate io CHILL simple built-in routine call	see section 7.4.3
implementation built-in routine call	see section 6.7
implementation location built-in routine call	see section 4.2.12
implementation value built-in routine call	see section 5.2.13
isassociated io CHILL value built-in routine call	see section 7.4.2
modification io CHILL simple built-in routine call	see section 7.4.5
readrecord io CHILL value built-in routine call	see section 7.4.9
writerecord io CHILL simple built-in routine call	see section 7.4.9

10.1.4.5 Miscellaneous semantic categories

array mode:	a *mode* in which the *composite mode* is an array mode.
discrete mode:	a *mode* in which the *non-composite mode* is a discrete mode.
location procedure call:	see section 4.2.11
modulion defining occurrence:	a *defining occurrence* which defines a *module name* or a *region name*.
non-apostrophe character:	a *character* which is not an apostrophe.
string mode:	a *mode* in which the *composite mode* is a string mode.
value procedure call:	see section 5.2.12

10.2 VISIBILITY AND NAME BINDING

10.2.1 Degrees of visibility

The binding rules are based on the visibility of *name strings* in the reaches of a program. Within a reach, each *name string* has one of the following four degrees of visibility:

Table 1. Degrees of visibility

Visibility	**Properties** (informal)
directly strongly visible	*Name string* is visible by creation, granting, seizing, or direct pervasiveness
indirectly strongly visible	*Name string* is inherited via block nesting or by its pervasive attribute
weakly visible	*Name string* is implied by a strongly visible *name string*
invisible	*Name string* may not be applied

A *name string* is said to be **strongly visible** in a reach if it is either **directly strongly visible** or **indirectly strongly visible** in that reach. A *name string* is said to be **visible** if it is either **weakly** or **strongly visible**, in that reach. Otherwise the *name string* is said to be **invisible** in that reach. The program structuring statements and visibility statements determine uniquely to which visibility class each *name string* belongs.

When a *name string* is visible in a reach, it can be **directly linked** to another *name string* in another reach, or **directly linked** to a *defining occurrence* in the program. The rules for direct linkage are in section 10.2.4.

Based on direct linkage, the notion of (not necessarily direct) linkage is defined as follows:

A *name string* N1, visible in reach R1, is said to be linked to *name string* N2 in reach R2 or to *defining occurrence* D, if and only if one of the following conditions holds:

- N1 in R1 is directly linked to N2 in R2 or to D
- N1 in R1 is directly linked to some N in some R, and N in R is linked to N2 in R2 or to D.

Note that linkage must be interpreted with respect to the visibility degree: a *name string* may be strongly visible in a reach with some linkages, and weakly visible in the same reach with other linkages.

10.2.2 Visibility conditions and name binding

In each reach of a program, the following conditions must be satisfied:

- Each *name string* visible in that reach must be linked to at least one *defining occurrence*.
- If a *name string* is strongly visible in a reach, and is linked to more than one *defining occurrence*, then all such *defining occurrences* (that are not quasi *defining occurrences*, for which rules are in 8.10) must be *defining occurrences* of **compatible** classes (in other words: must define the same set element), and all of them must be directly enclosed in one and the same reach.

A *name string* weakly visible in a reach, and linked as a weakly visible *name string* in that reach to *defining occurrences* that are not in **compatible** classes, is said to have a weak clash in that reach.

A *name string* NS, visible in reach R, is said to be bound in R to several *defining occurrences* according to the following rules:

- If NS is strongly visible in R, NS is bound to the *defining occurrences* to which it is linked in R (as a strongly visible *name string*);
- else, if NS is weakly visible in R, it is bound to the *defining occurrences* to which it is linked in R (as a weakly visible name string), provided NS has no weak clash in R. (Weak clashes are allowed in a reach if no *name* with a *name string* with a weak clash exists in the reach);
- otherwise, NS is not bound in R.

static condition: The *name string* attached to each name directly enclosed in a reach must be visible and bound in that reach.

binding of names: A name N with attached *name string* NS in a reach R is bound to the *defining occurrences* to which NS is bound in R.

10.2.3 Implied name strings

Each *name string* strongly visible in a reach R has a set of implied *name strings*, which are weakly visible in R, with the linkages specified below.

Each mode has a possibly empty set of implied *defining* occurrences attached in a reach, as listed in Table 2.

Each *name string* NS, strongly visible in reach R, has a set of implied *defining occurrences*, defined as follows, where D is one of the *defining occurrences* to which NS is bound in R:

- If D defines an **access** name of mode M, the implied *defining occurrences* of NS in R are those implied in R by M.
- If D defines a **mode** name, the implied *defining occurrences* of NS in R are those implied in R by the defining mode of the mode *name*.
- If D defines a **procedure** name, the implied *defining occurrences* of NS in R are those implied in R by the *parameter list* and of the *result spec* of the procedure, if any.
- If D defines a **signal** name, the implied *defining occurrences* of NS in R are all *defining occurrences* implied in R by all modes attached to the signal.
- If D defines a **process** name, the implied *defining occurrences* of NS in R are those implied in R by the *parameter list*, if any.

Table 2. Implied defining occurrences of modes in reach R

Modes	Set of implied defining occurrences
INT , *BIN* , *CHAR* *INSTANCE* , *PTR* *BOOL* , **EVENT** *CHAR (n)*, *BIN (n)* *BIT (n)*, **RANGE** *(...)* *ASSOCIATION*	Empty
mode name	The set of defining occurrences implied in R by its defining mode
M(m:n)	The set of defining occurrences implied in R by *M*
mode name (...) (parameterised)	The set of defining occurrences implied in R by *mode name*
REF *M*, **ROW** *M* **READ** *M*, **POWERSET** *M* **BUFFER** *M*	The set of defining occurrences implied in R by *M*
SET *(...)*	The set of **set element** defining occurrences in the mode
PROC $(M_1, ..., M_n)(M_{n+1})$	The union of the sets of the defining occurrences implied in R by M_1 through M_{n+1}
ACCESS *(M) N*	The union of the sets of the defining occurrences implied in R by *M*, *N*, *USAGE* and *WHERE*
ARRAY *(M) N*	The union of the sets of the defining occurrences implied in R by *M* and *N*
STRUCT $(N_1\ M_1, ..., N_n\ M_n)$	The union of the sets of defining occurrences implied in R by M_i for fields that are visible in R. For variant structures it is the union of the defining occurrences implied in R by the fields of the variant structure that are visible in R

If a *name string* NS, strongly visible in a reach R, has implied *defining occurrences*, each of those *defining occurrences* specifies an implied *name string* for NS in R: let D be a *defining occurrence* implied by NS in R and let Ni be the *name string* of D. There are two cases:

- NS is a *simple name string*. Then Ni directly linked in R to D is an implied *name string* of NS.
- NS is of the form P ! S, where S is a *simple name string*. Then P ! Ni directly linked in R to D is an implied *name string* of NS.

examples:

```
m: MODULE
       DCL x SET (on, off);
       GRANT x PREFIXED ;
   END ;
/* m ! x visible here with implied m ! on, m ! off */
```

10.2.4 Visibility in reaches

10.2.4.1 General

A *name string* is directly strongly visible in a reach according to the following rules :

- the *name string* is seized into the reach (see 10.2.4.5);

- the *name string* is granted into the reach (see 10.2.4.4);
- there is a *defining occurrence* with that *name string* in the reach. In that case, the *name string* in the reach is directly linked to the *defining occurrence*. (Note that the *name string* may be directly linked to several *defining occurrences* in the reach.)
- the *name string* is strongly visible in a directly enclosing reach and has the directly pervasive property in that reach. In that case, the *name string* also has the directly pervasive property in the reach, and is directly linked to the same *name string* in the directly enclosing reach.

A *name string* is indirectly strongly visible in a reach according to the following rules :

- The reach is a block in which the *name string* is not directly strongly visible, and the *name string* is strongly visible in the directly enclosing reach. The *name string* is said to be inherited by the block, and is directly linked to the same *name string* in the directly enclosing reach.
- The *name string* is not directly strongly visible in the reach, and the *name string* is strongly visible in a directly enclosing reach, where it has the pervasive property. The *name string* in the reach is directly linked to the *name string* in the directly enclosing reach. The *name string* has the pervasive property in the reach.
- The *name string* is a language or implementation defined *name string*, and the reach is a *context* with no *context* in front of it or the imaginary outermost *process definition*. The *name string* is considered to be directly linked to a *defining occurrence* for its predefined meaning. The *name string* has the pervasive property.

A *name string* is weakly visible in a reach if it is implied by a *name string* which is strongly visible in the reach, the rules for linkage are in 10.2.3.

10.2.4.2 Visibility statements

syntax:

<visibility statement> ::= *(1)*
<grant statement> *(1.1)*
| *<seize statement>* *(1.2)*

semantics: Visibility statements are only allowed in modulion reaches and control the visibility of the name strings mentioned in them and implicitly of their implied name strings.

static properties: A *visibility statement* has one or two origin reaches (see 8.2) and one or two destination reaches attached, defined as follows:

- If the *visibility statement* is a *seize statement*, its destination reach is the modulion reach directly enclosing the *seize statement*, and its origin reaches are the reaches directly enclosing that modulion reach.
- If the *visibility statement* is a *grant statement*, then its origin reach is the modulion reach directly enclosing the *grant statement*, and its destination reaches are the reaches directly enclosing that modulion reach.

10.2.4.3 Prefix rename clause

syntax:

<prefix rename clause> ::= *(1)*
(<old prefix> -> <new prefix>) ! <postfix> *(1.1)*

<old prefix> ::= *(2)*
<prefix> *(2.1)*
| *<empty>* *(2.2)*

<new prefix> ::= *(3)*
<prefix> *(3.1)*

| <*empty*> (3.2)

<*postfix*> ::= (4)
<*seize postfix*> {,<*seize postfix*>}* (4.1)
| <*grant postfix*> {,<*grant postfix*>}* (4.2)

derived syntax: A *prefix rename clause* where the *postfix* consists of more than one *seize postfix* (*grant postfix*) is derived syntax for several *prefix rename clauses*, one for each *seize postfix* (*grant postfix*), separated by commas, with the same *old prefix* and *new prefix*.

For example :

GRANT *(p -> q) ! a , b ;*

is derived syntax for

GRANT *(p -> q) ! a , (p -> q) ! b ;*

semantics: Prefix rename clauses are used in visibility statements to express change of prefix in prefixed name strings that are granted or seized. (Since prefix rename clauses can be used without prefix changes –when both the old prefix and the new prefix are empty– they are taken as the semantic base for visibility statements).

static properties: A *prefix rename clause* has one or two origin reaches attached, which are the origin reaches of the *visibility statement* in which it is written.

A *prefix rename clause* has one or two destination reaches attached, which are the destination reaches of the *visibility statement* in which it is written.

A *postfix* has a set of *name strings* attached, which is the set of *name strings* attached to its *seize postfix* or the set of name strings attached to its *grant postfix*. These *name strings* are the postfix *name strings* of the *prefix rename clause*.

A *prefix rename clause* has a set of old *name strings* and a set of new *name strings* attached. Each postfix *name string* attached to the *prefix rename clause* gives both an old *name string* and a new *name string* attached to the *prefix rename clause*, as follows: the new *name string* is obtained by prefixing the postfix *name string* with the *new prefix*; the old *name string* is obtained by prefixing the postfix *name string* with the *old prefix*.

When a new *name string* and an old *name string* are obtained from the same postfix *name string*, the old *name string* is said to be the source of the new *name string*.

visibility rules: The new *name strings* attached to a *prefix rename clause* are strongly visible in their destination reaches and are linked in those reaches to their sources in the origin reaches. If the *prefix rename clause* is part of a *seize* (*grant*) *statement*, those *name strings* are seized (granted) in their destination reach (reaches).

A *name string* NS strongly visible in reach R is said to be seizable by modulion M directly enclosed in R if NS is not linked in R to any *name string* in the reach of M.

A *name string* NS weakly visible in reach R is said to be seizable by modulion M directly enclosed in R if NS is linked in R to a *defining occurrence* not surrounded by the reach of M.

A *name string* NS strongly visible in reach R of modulion M is said to be grantable by M if NS is not linked in R to NS in the reach directly enclosing M.

A *name string* NS weakly visible in reach R of modulion M is said to be grantable by M if NS is linked in R to a *defining occurrence* surrounded by R.

static conditions: If a *prefix rename clause* is in a *seize statement* directly enclosed in the reach of modulion M then each of its old *name strings* must be :

- visible and bound in the reach directly enclosing the reach of M and

- seizable by M.

If a *prefix rename clause* is in a *grant statement* directly enclosed in the reach of modulion M then each of its old *name strings* must be :

- visible and bound in the reach of M and
- grantable by M.

If a modulion reach contains several *prefix rename clauses* in *seize* (*grant*) *statement*(s) then their sets of new *name strings* must be two by two disjoint.

examples:

25.35 *(stack ! int -> stack)!* **ALL** *(1.1)*

10.2.4.4 Grant statement

syntax:

<grant statement> ::= *(1)*
GRANT *<prefix rename clause>* {,*<prefix rename clause>*}*
[[**DIRECTLY**] **PERVASIVE**] ; *(1.1)*
| **GRANT** *<grant window>* [*<prefix clause>*]
[[**DIRECTLY**] **PERVASIVE**] ; *(1.2)*

<grant window> ::= *(2)*
<grant postfix> { , *<grant postfix>* }* *(2.1)*

<grant postfix> ::= *(3)*
<name string> *(3.1)*
| *<newmode name string>* *<forbid clause>* *(3.2)*
| [*<prefix>* *!*] **ALL** *(3.3)*

<prefix clause> ::= *(4)*
PREFIXED [*<prefix>*] *(4.1)*

<forbid clause> ::= *(5)*
FORBID { *<forbid name list>* | **ALL** } *(5.1)*

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

semantics: Grant statements are a means of extending the visibility of name strings in a modulion reach into the directly enclosing reaches. **FORBID** can be specified only for newmode names which are structure modes. It means that all locations and values of that mode have fields which may be selected only inside the granting modulion, not outside.

The following visibility rules apply:

- If the *grant statement* contains *prefix rename clause*(s), the *grant statement* has the effect of its *prefix rename clause*(s) (see 10.2.4.3).
- If the *grant statement* contains *grant windows*, it is shorthand notation for a set of *grant statements* with *prefix rename clauses* constructed as follows:
  - There is a *grant statement* for each *grant postfix* in the *grant window*.
  - The *old prefix* in their *prefix rename clause* is empty.
  - The *new prefix* in their *prefix rename clause* is the *prefix* attached to the *prefix clause* in the *grant statement*, or it is empty if there is no *prefix clause* in the original *grant statement*.

- The *postfix* in the *prefix rename clause* is the corresponding *postfix* in the *grant window*.

- The notation **FORBID ALL** is a syntactic shorthand forbidding all the *field names* of the **newmode** name (see section 10.2.5).

- If a *prefix rename clause* in a *grant statement* has a *grant postfix* which contains a *prefix* and **ALL** , then it is of the form

  (OP->NP) ! P ! **ALL**

  where *OP* and *NP* are the possibly empty *old prefix* and *new prefix* respectively and *P* is the *prefix* in the *grant postfix*. The *prefix rename clause* is then equivalent to a clause of the form

  (OP ! P->NP ! P) ! **ALL**

static properties: When a *grant statement* contains (**DIRECTLY**) **PERVASIVE** , then all *name strings* granted by it have the (directly) pervasive property in the surrounding reaches of the modulion in which the *grant statement* is contained.

A *prefix clause* has a *prefix* attached, defined as follows:

- If the *prefix clause* contains a *prefix*, then that *prefix* is attached.

- Otherwise, the attached *prefix* is a *simple prefix* whose name string is determined as follows:

  - If the reach directly enclosing the *prefix* is a *module*, *region*, *quasi module* or *quasi region*, then the name string is the same as the one of the modulion name of that modulion.

  - If the reach directly enclosing the *prefix* is a *spec region* or *spec module*, then the name string is the name string in front of **SPEC** .

A *grant postfix* has a set of *name strings* attached, defined as follows:

- If it is a *name string*, or contains a **newmode** *name string*, then the set containing only that *name string*.

- Otherwise, let *OP* be the (possibly empty) *old prefix* of the *prefix rename clause* in which the *grant postfix* is placed, the set contains all *name strings* of the form *OP ! N* (i.e., obtained by prefixing *N* with *OP*) for any *name string N* such that *OP ! N* is strongly visible in the reach of the modulion in which the *grant postfix* is placed and grantable by this modulion.

static conditions: The **newmode** *name string* with forbid specification must be a *simple name string* and must be strongly visible in the reach *R* of the modulion in which the *grant statement* is placed. The **newmode** *name string* must be bound in *R* to the *defining occurrence* of a newmode, which must be a structure mode, and each *field name* in the *field name list* must be a *field name* of that mode. The newmode *defining occurrence* must be directly enclosed in *R*. All **field** names in a *forbid name list* must have different name strings.

If the *grant statement* is placed in the reach of a *region* or *spec region*, it must not grant a *name string* which is bound in the reach of the *region* or *spec region* to **intra-regional** *defining occurrences*.

The *prefix rename clause* in a *grant statement* must have a *grant postfix*.

A *name string* granted by a *grant statement* in the reach of a *spec module* or a *spec region* must be bound in that reach to a *defining occurrence* surrounded by that reach. That *defining occurrence* may not have a *quasi module* or *quasi region* attached.

If a *grant statement* contains a *prefix clause* which does not contain a *prefix*, then its directly enclosing modulion must not be a *context* and,

- if its directly enclosing modulion is a *module*, *region*, *quasi module* or *quasi region*, then it must be named (i.e., it must be headed by a *defining occurrence* followed by a colon);

- if its directly enclosing modulion is a *spec module* or a *spec region*, then it must be headed by a *simple name string*.

examples:

25.10 **GRANT ALL PREFIXED** *stack ! char ;* *(1.1)*
6.44 *gregorian_date, julian_day_number* *(2.1)*

10.2.4.5 Seize statement

syntax:

<seize statement> ::= *(1)*
SEIZE *<prefix rename clause>* { , *<prefix rename clause>*}*; *(1.1)*
| **SEIZE** *<seize window>* [*<prefix clause>*] ; *(1.2)*

<seize window> ::= *(2)*
<seize postfix> { , *<seize postfix>* }* *(2.1)*

<seize postfix> ::= *(3)*
<name string> *(3.1)*
| *<modulion name string>* **ALL** *(3.2)*
| [*<prefix>* !] **ALL** *(3.3)*

<modulion name string> ::= *(4)*
<modulion name string> *(4.1)*

semantics: Seize statements are a means of extending the visibility of name strings in group reaches into the reaches of directly enclosed modulions.

The following visibility rules apply:

- If a *name string* which has the (directly) pervasive property in the directly enclosing reaches is seized, it will be **directly strongly visible** in the reach of the seizing modulion and it keeps the (directly) pervasive property.

- If the *seize statement* contains *prefix rename clause*(s), the *seize statement* has the effect of its *prefix rename clause*(s) (see 10.2.4.3).

- If the *seize statement* contains a *seize window*, it is shorthand notation for a set of *seize statements* with *prefix rename clauses* constructed as follows:

  - For each *seize postfix* in the *seize window*, there is a corresponding *seize statement*.

  - The *old prefix* in their *prefix rename clause* is the *prefix* attached to the *prefix clause* in the *seize statement*, or is empty if there is no *prefix clause* in the original *seize statement*.

  - The *new prefix* in their *prefix rename clause* is empty.

  - The *postfix* in their *prefix rename clause* is the corresponding *postfix* of the *seize window*.

- If a *prefix rename clause* in a *seize statement* has a *seize postfix* which contains a *prefix* and **ALL** , then it is of the form

  (OP->NP) ! P ! **ALL**

  where OP and NP are the possibly empty *old prefix* and *new prefix* respectively and P is the *prefix* in the *seize postfix*. The *prefix rename clause* is then equivalent to a clause of the form

(OP ! P->NP ! P) ! **ALL**

- If a *prefix rename clause* in a *seize statement* has a *seize postfix* which contains a modulion *name string* and **ALL** , then it is of the form

  (OP->NP) ! MN **ALL**

  where *OP* and *NP* are the possibly empty *old prefix* and *new prefix* respectively and *MN* is the modulion *name string* in the *seize postfix*. The *prefix rename clause* is then equivalent to a set of *seize statements*, each one with a *prefix rename clause* of the of the form

  (OP->NP) ! NS

  with a *seize statement* for each *name string NS* such that:

  - *NS* is strongly visible in the reach that directly encloses the modulion directly enclosing the *seize statement* and is seizable by this modulion;
  - *NS* is granted by the modulion attached to the *defining occurrence* to which *MN* is bound in the reach directly enclosing the modulion in which the *seize statement* is placed.

static properties: A *seize postfix* has a set of *name strings* attached, defined as follows:

- If the *seize postfix* is a *name string*, the set containing only the *name string*.
- Else, if the *seize postfix* is **ALL** , let *OP* be the (possibly empty) *old prefix* of the *prefix rename clause* of which the *seize postfix* is part, the set contains all *name strings* of the form *OP ! S*, for any *name string S*, such that *OP ! S* is strongly visible in the reach directly enclosing the modulion in which the *seize statement* is placed and seizable by this modulion.

static conditions: A *name string* in the set of old *name strings* attached to a *prefix rename clause* in a *seize statement* must not be bound to a **value do with** *defining occurrence* nor a **location do with** *defining occurrence* in a reach which directly encloses the modulion in which the *visibility statement* is placed.

The *prefix rename clause* in a *seize statement* must have a *seize postfix*.

If a *seize statement* contains a *prefix clause* which does not contain a *prefix*, then its directly enclosing modulion must not be a *context* and,

- if its directly enclosing modulion is a *module*, *region*, *quasi module* or *quasi region*, then it must be named (i.e., it must be headed by a *defining occurrence* followed by a colon);
- if its directly enclosing modulion is a *spec module* or a *spec region*, then it must be headed by a *simple name string*.

The **modulion** *name string* in a *modulion name string* must be bound, in reaches directly enclosing the reach in which the *modulion name string* is placed, to a **modulion** *defining occurrence*. It must not be bound to a *defining occurrence* to which a *quasi region* or *quasi module* is attached.

examples:

25.35 **SEIZE** *(stack ! int -> stack) !* **ALL** *;* *(1.1)*
25.26 **SEIZE ALL PREFIXED** *stack ;* *(1.2)*

10.2.5 Visibility of field names

Field names may occur only in the following contexts:

- *Field selection* in *structure field* and *value structure field.*
- *Labelled structure tuples.*
- *Forbid clauses* in the *grant statement.*

In each of these cases, the *name string* of the *field name* can be bound to a *field name defining occurrence* in the mode M or in the defining mode of M, obtained as follows:

- M is the mode of the **structure** *location* or (**strong**) **structure** *value*.
- M is the mode of the *structure tuple*.
- M is the mode of the *defining occurrence* to which the **newmode** *name string* is bound in the reach in which the *forbid clause* is placed.

However, if the **novelty** of M is a **newmode** *name* which has been granted by a modulion with a *forbid clause*, then outside the granting modulion the *field names* mentioned in the *forbid name list* are invisible and cannot be used.

11 EXCEPTION HANDLING

11.1 GENERAL

An exception is either a language defined exception, in which case it may have a language defined exception name, a user defined exception, or an implementation defined exception. A language defined exception will be caused by the dynamic violation of a dynamic condition. Any exception can be caused by the execution of a cause action.

When an exception is caused, it may be handled, i.e., an action statement list of an appropriate handler will be executed.

Exception handling is defined such that at any statement it is statically known which exceptions might occur (i.e., it is statically known which exceptions cannot occur) and for which exceptions an appropriate handler can be found or which exceptions may be passed to the calling point of a procedure. If an exception occurs and no handler for it can be found, the program is in error.

11.2 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 exception 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 exception 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 **exception** names in all the *exception list* occurrences must be different.

dynamic conditions: The *SPACEFAIL* exception occurs if an action statement list is entered and storage requirements cannot be satisfied.

examples:

10.47 **ON**
(ALLOCATEFAIL): **CAUSE** *overflow;*
END *(1.1)*

11.3 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 an exception with exception name 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, a module or a region, the appropriate handler (if present) is the appropriate handler for the bracketed action, module or region with respect to E;

3. otherwise, if A or D is placed in the reach of a procedure definition then:

   - if a handler 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 data statement or region 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.

12 IMPLEMENTATION OPTIONS

12.1 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 that is not a **reserved** simple name string 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 that is considered to be defined in the reach of the imaginary outermost process definition or in any context (see section 8.8). 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).

12.2 IMPLEMENTATION DEFINED INTEGER MODES

An implementation may define integer modes other than the ones defined by *INT* ; e.g., short integers, long integers, unsigned integers. These integer modes must be denoted by implementation defined integer mode names. These names are considered to be **newmode** names, **similar** to *INT* . Their value ranges are implementation defined. These integer-modes may be defined as **root** modes of appropriate classes.

12.3 IMPLEMENTATION DEFINED REGISTER NAMES

An implementation may define a set of pre-defined **register** names (see sections 2.7 and 3.7).

12.4 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 imaginary outermost process or in any context. 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.

12.5 IMPLEMENTATION DEFINED HANDLERS

An implementation may specify that an implementation defined handler is appended to the imaginary outermost process definition (see section 8.8). 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.

12.6 IMPLEMENTATION DEFINED REFERABILITY

An implementation may define other (sub-)locations to be **referable** in addition to the locations which are defined to be **referable** by the language (see section 4.2.1).

12.7 SYNTAX OPTIONS

At some places, CHILL allows for more than one syntactic description for the same semantics. The choice for one of the following options must be fixed within the whole program.

Assignment symbol

The assignment symbol is either := or =

ARRAY

The **reserved** simple name string **ARRAY** is either mandatory or not allowed.

RETURNS

In procedure definitions with a **result spec**, the **reserved** simple name string **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.

APPENDIX A: CHARACTER SETS FOR CHILL

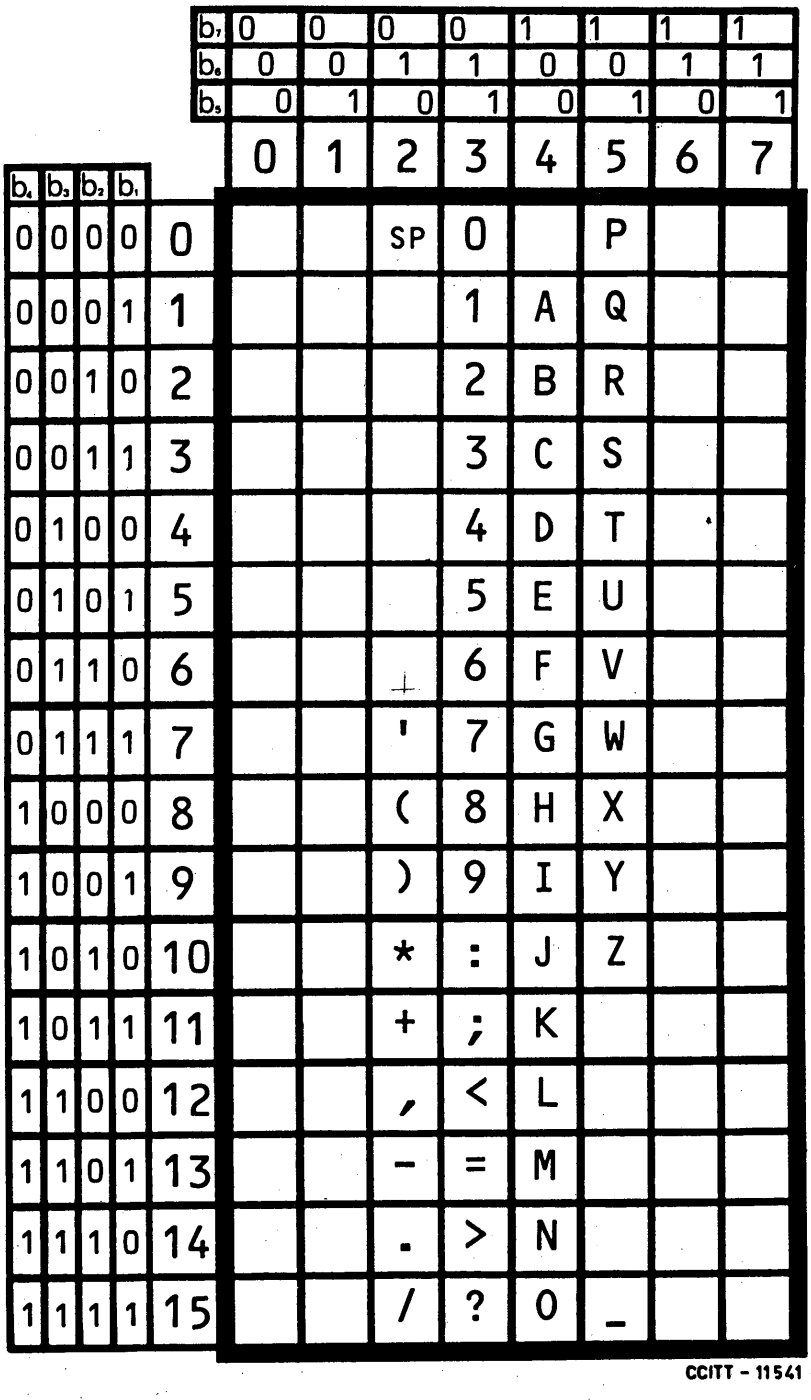
A.1 CCITT ALPHABET NO. 5 INTERNATIONAL REFERENCE VERSION

Recommendation V3 (The internal representation is the binary number formed by bits b7 to b1, where b1 is the least significant bit).

b_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
b_4 b_3 b_2 b_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

CCITT-11540

A.2 MINIMAL CHARACTER SET FOR CHILL



APPENDIX B: SPECIAL SYMBOLS AND CHARACTER COMBINATIONS

	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, definition indicator
/=	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, prefix renaming
<>	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
/.	prefixing operator	prefixing of names
!	prefixing operator	prefixing of names
B'	literal qualification	binary base for literal
D'	literal qualification	decimal base for literal
H'	literal qualification	hexadecimal base for literal
O'	literal qualification	octal base for literal
C'	literal qualification	hexadecimal representation for character string literal

APPENDIX C: SPECIAL SIMPLE NAME STRINGS

C.1 RESERVED SIMPLE NAME STRINGS

ACCESS
ADDR
ALL
ARRAY
ASSERT

BASED
BEGIN
BUFFER
BY

CALL
CASE
CAUSE
CONTEXT
CONTINUE

DCL
DELAY
DIRECTLY
DO
DOWN
DYNAMIC

ELSE
ELSIF
END
ENTRY
ESAC
EVENT
EVER
EXCEPTIONS
EXIT

FI
FOR
FORBID

GENERAL
GOTO
GRANT

IF
IN
INIT
INLINE
INOUT

LOC

MODULE

NEWMODE
NONREF
NOPACK

OD
OF
ON
OUT

PACK
PERVASIVE
POS
POWERSET
PREFIXED
PRIORITY
PROC
PROCESS

RANGE
READ
RECEIVE
RECURSIVE
REF
REGION
REMOTE
RESULT
RETURN
RETURNS
ROW

SEIZE
SEND
SET
SIGNAL
SIMPLE
SPEC
START
STATIC
STEP
STOP
STRUCT
SYN
SYNMODE

THEN
TO

UP

WHILE
WITH

C.2 PREDEFINED SIMPLE NAME STRINGS

ABS
AND
ALLOCATE
ASSOCIATE
ASSOCIATION

BIN
BIT
BOOL

CARD
CHAR
CONNECT
CREATE

DELETE
DISCONNECT
DISSOCIATE

EXISTING
FALSE
FIRST

GETASSOCIATION
GETSTACK
GETUSAGE

INDEXABLE
INSTANCE
INT
ISASSOCIATED

LAST
LOWER

MAX
MIN
MOD
MODIFY

NOT
NULL
NUM

OR
OUTOFFILE

PRED
PTR

READABLE
READONLY
READRECORD
READWRITE
REM

SEQUENCIBLE
SAME
SIZE
SUCC

TERMINATE
THIS
TRUE

UPPER
USAGE

VARYING

WHERE
WRITEABLE
WRITEONLY
WRITERECORD

XOR

C.3 EXCEPTION NAMES

ALLOCATEFAIL
ASSERTFAIL
ASSOCIATEFAIL
CONNECTFAIL
CREATEFAIL
DELAYFAIL
DELETEFAIL
EMPTY
EXTINCT
MODIFYFAIL
NOTCONNECTED
NOTASSOCIATED
OVERFLOW
RANGEFAIL
READFAIL
RECURSEFAIL
SENDFAIL
SPACEFAIL
TAGFAIL
TERMINATEFAIL
WRITEFAIL

C.4 DIRECTIVES

FREE

APPENDIX D: PROGRAM EXAMPLES

1. **Operations on integers**

```
1   integer_operations:
2    MODULE
3
4        add:
5        PROC (i,j INT )( INT ) EXCEPTIONS (OVERFLOW);
6                RESULT i+j;
7        END add;
8
9        mult:
10       PROC (i,j INT )( INT ) EXCEPTIONS (OVERFLOW);
11               RESULT i*j;
12       END mult;
13
14       GRANT add, mult;
15       SYNMODE operand_mode= INT ;
16       GRANT operand_mode;
17       SYN neutral_for_add=0,
18           neutral_for_mult=1;
19       GRANT neutral_for_add,
20                 neutral_for_mult;
21
22   END integer_operations;
```

2. **Same operations on fractions**

```
1   fraction_operations:
2    MODULE
3        NEWMODE fraction= STRUCT (num,denum INT );
4
5        add:
6        PROC (f1,f2 fraction)(fraction) EXCEPTIONS (OVERFLOW);
7                RETURN [f1.num*f2.denum+f2.num*f1.denum,f1.denum*f2.denum];
8        END add;
9
10       mult:
11       PROC (f1,f2 fraction)(fraction) EXCEPTIONS (OVERFLOW);
12               RETURN [f1.num*f2.num,f2.denum*f1.denum];
13       END mult;
14
15       GRANT add, mult;
16       SYNMODE operand_mode=fraction;
17       GRANT operand_mode;
18       SYN neutral_for_add fraction=[ 0,1 ],
19           neutral_for_mult fraction=[ 1,1 ];
20       GRANT neutral_for_add,
21                 neutral_for_mult;
22
23   END fraction_operations;
```

3. **Same operations on complex numbers**

```
1   complex_operations:
2    MODULE
3        NEWMODE complex= STRUCT (re,im INT );
4
5        add:
6        PROC (c1,c2 complex)(complex) EXCEPTIONS (OVERFLOW);
7                RETURN [c1.re+c2.re,c1.im+c2.im];
8        END add;
9
10       mult:
11       PROC (c1,c2 complex)(complex) EXCEPTIONS (OVERFLOW);
12               RETURN [c1.re*c2.re-c1.im*c2.im,c1.re*c2.im+c1.im*c2.re];
13       END mult;
14
15       GRANT add, mult;
16       SYNMODE operand_mode=complex;
17       GRANT operand_mode;
18       SYN neutral_for_add=complex [ 0,0 ],
19             neutral_for_mult=complex [ 1,0 ];
20       GRANT neutral_for_add,
21                neutral_for_mult;
22
23   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 INT INOUT , b,c,order INT )
5                EXCEPTIONS (wrong_input) RECURSIVE ;
6            DCL d INT ;
7            ASSERT b>0 AND c>0 AND order>0
8                ON (ASSERTFAIL):
9                    CAUSE wrong_input;
10               END ;
11           CASE order OF
12               (1):        a := b+c;
13                            RETURN ;
14               (2):        d := 0;
15               ( ELSE ): d := 1;
16           ESAC ;
17           DO FOR i := 1 TO c;
18               op (a,b,d,order-1);
19               d := a;
20           OD ;
21           RETURN ;
22       END op;
23
24       GRANT op;
25
26   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           DCL c STRUCT (c4,c2,c1 BOOL );
7           DCL k2,x,w,t,s,r BOOL ;
8           DO WITH a,b,c;
9              k2 := a1 AND b1;
10             c1 := NOT k2 AND (a1 OR b1);
11             x := a2 AND b2 AND k2;
12             w := a2 OR b2 OR k2;
13             t := b2 AND k2;
14             s := a2 AND k2;
15             r := a2 AND b2;
16             c4 := r OR s OR t;
17             c2 := x OR (w AND NOT c4);
18          OD ;
19          RETURN c;
20       END adder;
21       GRANT adder;
22    END add_bit_by_bit;
23
24  exhaustive_checker:
25    MODULE
26       SEIZE adder;
27       DCL a STRUCT (a2,a1 BOOL ),
28             b STRUCT (b2,b1 BOOL );
29       SYNMODE res= ARRAY (1:16) STRUCT (c4,c2,c1 BOOL );
30       DCL r INT , results res;
31       DO WITH a,b;
32          r := 0;
33           DO FOR a2 IN BOOL ;
34              DO FOR a1 IN BOOL ;
35                 DO FOR b2 IN BOOL ;
36                    DO FOR b1 IN BOOL ;
37                       r+ := 1;
38                       results (r) := adder (a,b);
39                    OD ;
40                 OD ;
41              OD ;
42           OD ;
43       OD ;
44       ASSERT
45          results=res [[ FALSE , FALSE , FALSE ],[ FALSE , FALSE , TRUE ],
46                       [ FALSE , TRUE , FALSE ] ,[ FALSE , TRUE , TRUE ],
47                       [ FALSE , FALSE , TRUE ] ,[ FALSE , TRUE , FALSE ],
48                       [ FALSE , TRUE , TRUE ] ,[ TRUE , FALSE , FALSE ],
49                       [ FALSE , TRUE , FALSE ] ,[ FALSE , TRUE , TRUE ],
50                       [ TRUE , FALSE , FALSE ] ,[ TRUE , FALSE , TRUE ],
51                       [ FALSE , TRUE , TRUE ] ,[ TRUE , FALSE , FALSE ],
52                       [ TRUE , FALSE , TRUE ] ,[ TRUE , TRUE , FALSE ]];
53    END exhaustive_checker;
```

6. Playing with dates

```
1   playing_with_dates:
2    MODULE /* from collected algorithms from CACM no. 199 */
3        SYNMODE month= SET (jan,feb,mar,apr,may,jun,
4                    jul,aug,sep,oct,nov,dec);
5        NEWMODE date= STRUCT (day INT (1:31), mo month, year INT );
6
7        gregorian_date:
8        PROC (julian_day_number INT )(date);
9            DCL j INT := julian_day_number,
10                   d,m,y INT ;
11           j := 1_721_119;
12           y := (4 * j - 1) / 146_097;
13           j := 4 * j - 1 - 146_097 * y;
14           d := j / 4;
15           j := (4 * d + 3) / 1_461;
16           d := 4 * d + 3 - 1_461 * j;
17           d := (d + 4) / 4;
18           m := (5 * d - 3) / 153;
19           d := 5 * d - 3 - 153 * m;
20           d := (d + 5) / 5;
21           y := 100 * y + j;
22           IF m<100 THEN m + := 3;
23                    ELSE m - := 9;
24                         y + := 1;
25           FI ;
26           RETURN [d,month (m+1), y];
27       END gregorian_date;
28
29       julian_day_number:
30       PROC (d date)( INT );
31           DCL c,y,m INT ;
32           DO WITH d;
33               m := NUM (mo)+1;
34               IF m>2 THEN m - := 3;
35                      ELSE m + := 9;
36                           year - := 1;
37               FI ;
38               c := year/100;
39               y := year-100*c;
40               RETURN (146_097*c)/4+(1_461*y)/4
41                          +(153+m+c)/5+day+1_721_119;
42           OD ;
43       END julian_day_number;
44       GRANT gregorian_date, julian_day_number;
45   END playing_with_dates;
46
47   test:
48   MODULE
49       SEIZE gregorian_date, julian_day_number;
50       ASSERT julian_day_number ([ 10,dec,1979 ])=julian_day_number
51                (gregorian_date(julian_day_number([ 10,dec,1979 ])));
52   END test;
```

7. Roman numerals

```
1   Roman:
2    MODULE
3        SEIZE n,rn;
4        GRANT convert;
5       convert:
6        PROC () EXCEPTIONS (string_too_small);
7            DCL r INT := 0;
8            DO WHILE n>=1_000;
9               rn(r) := 'M';
10              n - := 1_000;
11              r + := 1;
12           OD ;
13           IF n>500 THEN rn(r) := 'D';
14                         n - := 500;
15                         r + := 1;
16           FI ;
17           DO WHILE n>=100;
18              rn(r) := 'C';
19              n - := 100;
20              r + := 1;
21           OD ;
22           IF n>=50 THEN rn(r) := 'L';
23                         n - := 50;
24                         r + := 1;
25           FI ;
26           DO WHILE n>=10;
27              rn(r) := 'X';
28              n - := 10;
29              r + := 1;
30           OD ;
31           IF n>=5 THEN rn(r) := 'V';
32                       n - := 5;
33                       r + := 1;
34           FI ;
35           DO WHILE n>=1;
36              rn(r) := 'I';
37              n - := 1;
38              r + := 1;
39           OD ;
40           RETURN ;
41       END ON (RANGEFAIL): DO FOR i := 0 TO UPPER (rn);
42                                 rn(i) := '.';
43                           OD ;
44                           CAUSE string_too_small;
45       END convert;
46   END Roman;
47  test:
48   MODULE
49       SEIZE convert;
50       DCL n INT INIT := 1979;
51       DCL rn CHAR (20) INIT := (20)' ';
52       GRANT n,rn;
53      convert ();
54       ASSERT rn='MDCCCCLXXVIIII'//(6)' ';
55   END test;
```

8. **Counting letters in a character string of arbitrary length**

```
letter_count:
 MODULE
     SEIZE max;
     DCL letter POWERSET CHAR INIT := ['A' : 'Z'];
     count:
     PROC (input ROW CHAR (max) IN , output ARRAY ('A':'Z') INT OUT );
         DO FOR i := 0 TO UPPER (input ->);
             IF input -> (i) IN letter
                 THEN
                     output (input -> (i)) + := 1;
             FI ;
         OD ;
     END count;
     GRANT count;
 END letter_count;
test:
 MODULE
     SYNMODE results= ARRAY ('A':'Z') INT ;
     DCL c CHAR (10) INIT := 'A-B<ZAA9K' ' ';
     DCL output results;
     SYN max=10_000;
     GRANT max;
     SEIZE count;
     count (-> c,output);
     ASSERT output=results [('A') : 3,('B','K','Z') : 1, ( ELSE ) : 0];
 END test;
```

9. **Prime numbers**

```
prime:
 MODULE

     SYN max = H'7FFF;
     NEWMODE number_list = POWERSET INT (2:max);
     SYN empty = number_list [ ];
     DCL sieve number_list INIT := [ 2:max ],
           primes number_list INIT := empty;
     GRANT primes;
     DO WHILE sieve/=empty;
           primes OR := [ MIN (sieve)];
           DO FOR j := MIN (sieve) BY MIN (sieve) TO max;
               sieve - := [j];
           OD ;
     OD ;
 END prime;
```

10. **Implementing stacks in two different ways, transparent to the user**

```
1   stack : MODULE
2       NEWMODE element = STRUCT (a INT , b BOOL );
3       stacks_1:
4       MODULE
5           SEIZE element;
6           SYN max=10_000,min=1;
7           DCL stack ARRAY (min : max) element,
8                 stackindex INT INIT := min;
9
10          push:
11          PROC (e element) EXCEPTIONS (overflow);
12                IF stackindex=max
13                    THEN CAUSE overflow;
14                FI ;
15                stackindex + := 1;
16                stack (stackindex) := e;
17                RETURN ;
18          END push;
19
20          pop:
21          PROC () EXCEPTIONS (underflow);
22                IF stackindex=min
23                    THEN CAUSE underflow;
24                FI ;
25                stackindex - := 1;
26                RETURN ;
27          END pop;
28
29          elem:
30          PROC (i INT )(element LOC ) EXCEPTIONS (bounds);
31                IF i<min OR i>max
32                    THEN CAUSE bounds;
33                FI ;
34                RETURN stack (i);
35          END elem;
36
37          GRANT push,pop,elem;
38      END stacks_1;
39      stacks_2:
40      MODULE
41          SEIZE element;
42          NEWMODE cell= STRUCT (pred,succ REF cell,info element);
43          DCL p,last,first REF cell INIT := NULL ;
44
45          push:
46          PROC (e element) EXCEPTIONS (overflow);
47              p := ALLOCATE (cell) ON
48                                  (ALLOCATEFAIL) : CAUSE overflow;
49                                   END ;
50              IF last= NULL
51                  THEN first := p;
52                       last := p;
53                  ELSE last ->. succ := p;
54                       p ->. pred := last;
55                       last := p;
56              FI ;
57              last ->. info := e;
58              RETURN ;
```

```
59          END push;
60
61          pop:
62          PROC () EXCEPTIONS (underflow);
63              IF last= NULL
64                  THEN CAUSE underflow;
65              FI ;
66              p := last;
67              last := last ->. pred;
68              IF last = NULL
69                  THEN first := NULL ;
70                  ELSE last ->. succ := NULL ;
71              FI ;
72              TERMINATE(p);
73              RETURN ;
74          END pop;
75
76          elem:
77          PROC (i INT ) (element LOC ) EXCEPTIONS (bounds);
78              IF first= NULL
79                  THEN CAUSE bounds;
80              FI ;
81              p := first;
82              DO FOR j := 2 TO i;
83                  IF p ->. succ= NULL
84                      THEN CAUSE bounds;
85                  FI ;
86                  p := p ->. succ;
87              OD ;
88              RETURN p ->. info;
89          END elem;
90
91          /* GRANT push,pop,elem; */
92      END stacks_2;
93  END stack;
```

11. **Fragment for playing chess**

```
chess_fragments:
 MODULE
     NEWMODE piece= STRUCT (color SET (white,black),
                            kind SET (pawn,rook,knight,bishop,queen,king));
     NEWMODE column= SET (a,b,c,d,e,f,g,h);
     NEWMODE line= INT (1 : 8);
     NEWMODE square= STRUCT (status SET (occupied,free),
                             CASE status OF
                             (occupied) : p piece,
                             (free) :
                             ESAC );
     NEWMODE board= ARRAY (line) ARRAY (column) square;
     NEWMODE move= STRUCT (lin_1,lin_2 line,
                           col_1,col_2 column);

     initialise:
     PROC (bd board INOUT );
         bd := [ (1): [(a,h):     [.status: occupied, .p : [white,rook]],
                      (b,g):      [.status: occupied, .p : [white,knight]],
                      (c,f):      [.status: occupied, .p : [white,bishop]],
                      (d):        [.status: occupied, .p : [white,queen]],
                      (e):        [.status: occupied, .p : [white,king]]],
                (2): [( ELSE ):[.status: occupied, .p : [white,pawn]]],
                (3:6):[( ELSE ):[.status: free]],
                (7): [( ELSE ):[.status: occupied, .p : [black,pawn]]],
                (8): [(a,h):     [.status: occupied, .p : [black,rook]],
                      (b,g):      [.status: occupied, .p : [black,knight]],
                      (c,f):      [.status: occupied, .p : [black,bishop]],
                      (d):        [.status: occupied, .p : [black,queen]],
                      (e):        [.status: occupied, .p : [black,king]]]
             ];
         RETURN ;
     END initialise;
     register_move:
     PROC (b board LOC ,m move) EXCEPTIONS (illegal);
         DCL starting square LOC := b (m.lin_1)(m.col_1),
             arriving square LOC := b (m.lin_2)(m.col_2);
         DO WITH m;
             IF starting.status=free THEN CAUSE illegal; FI ;
             IF arriving.status/=free THEN
                 IF arriving.p.kind=king THEN CAUSE illegal; FI ;
             FI ;
             CASE starting.p.kind, starting.p.color OF
                 (pawn),(white):
                  IF col_1 = col_2 AND (arriving.status/=free
                      OR NOT (lin_2=lin_1+1 OR lin_2=lin_1+2 AND lin_2=2))
                      OR (col_2= PRED (col_1) OR col_2= SUCC (col_1))
                      AND arriving.status=free THEN CAUSE illegal; FI ;
                      IF arriving.status/=free THEN
                          IF arriving.p.color=white THEN CAUSE illegal; FI ; FI ;
                 (pawn),(black):
                  IF col_1=col_2 AND (arriving.status/=free
                      OR NOT (lin_2=lin_1-1 OR lin_2=lin_1-2 AND lin_1=7))
                      OR (col_2= PRED (col_1) OR col_2= SUCC (col_1))
                      AND arriving.status=free THEN CAUSE illegal; FI ;
                      IF arriving.status/=free THEN
                          IF arriving.p.color=black THEN CAUSE illegal; FI ; FI ;
                 (rook),(*):
```

```
59                  IF NOT ok_rook (b,m)
60                      THEN CAUSE illegal;
61                  FI ;
62                  (bishop),(*):
63                  IF NOT ok_bishop (b,m)
64                      THEN CAUSE illegal;
65                  FI ;
66                  (queen),(*):
67                  IF NOT ok_rook (b,m) AND NOT ok_bishop (b,m)
68                      THEN CAUSE illegal;
69                  FI ;
70                  (knight),(*):
71                  IF ABS ( ABS ( NUM (col_2)- NUM (col_1))
72                           - ABS (lin_2- lin_1)) /= 1
73                      OR ABS ( NUM (col_2)- NUM (col_1))
74                          + ABS (lin_2- lin_1) =/ 3 THEN CAUSE illegal; FI ;
75                  IF arriving.status/=free THEN
76                          IF arriving.p.color=starting.p.color THEN
77                          CAUSE illegal; FI ; FI ;
78                  (king),(*):
79                  IF ABS ( NUM (col_2)- NUM (col_1)) > 1
80                      OR ABS (lin_2- lin_1) > 1
81                      OR lin_2=lin_1 AND col_2=col_1 THEN CAUSE illegal; FI ;
82                  IF arriving.status/=free THEN
83                      IF arriving.p.color=starting.p.color THEN
84                      CAUSE illegal; FI ; FI ;/* checking king moving to check not implemented
*/
85              ESAC ;
86          OD ;
87          arriving := starting;
88          starting := [.status:free];
89          RETURN ;
90      END register_move;
91      ok_rook:
92      PROC (b board,m move)( BOOL );
93          DCL starting square := b (m.lin_1)(m.col_1),
94              arriving square := b (m.lin_2)(m.col_2);
95
96          DO WITH m;
97              IF NOT (col_2=col_1 OR lin_1=lin_2) THEN RETURN FALSE ; FI ;
98              IF arriving.status/=free THEN
99                  IF arriving.p.color=starting.p.color THEN ;
100                 RETURN FALSE ; FI ; FI ;
101             IF col_1=col_2
102                 THEN IF lin_1<lin_2
103                         THEN DO FOR lin := lin_1+1 TO lin_2-1;
104                                 IF b (lin)(col_1).status/=free
105                                     THEN RETURN FALSE ;
106                                 FI ;
107                             OD ;
108                         ELSE DO FOR lin := lin_1-1 DOWN TO lin_2+1;
109                                 IF b (lin)(col_1).status/=free
110                                     THEN RETURN FALSE ;
111                                 FI ;
112                             OD ;
113                     FI ;
114                 ELSIF col_1<col_2
115                         THEN DO FOR col := SUCC (col_1) TO PRED (col_2);
116                                 IF b (lin_1)(col).status/=free
117                                     THEN RETURN FALSE ;
118                                 FI ;
```

```
119                             OD ;
120                         ELSE DO FOR col := SUCC (col_2) DOWN TO PRED (col_1);
121                                 IF b (lin_1)(col).status/=free
122                                     THEN RETURN FALSE ;
123                                 FI ;
124                             OD ;
125             FI ;
126             RETURN TRUE ;
127         OD ;
128     END ok_rook;
129     ok_bishop:
130     PROC (b board,m move)( BOOL );
131         DCL starting square := b (m.lin_1)(m.col_1),
132             arriving square := b (m.lin_2)(m.col_2),
133             col column;
134
135         DO WITH m;
136             CASE lin_2>lin_1,col_2>col_1 OF
137                 ( TRUE ),( TRUE ): col := col_1;
138                         DO FOR lin := lin_1+1 TO lin_2-1;
139                                 col := SUCC (col);
140                                 IF b (lin)(col).status/=free
141                                     THEN RETURN FALSE ;
142                                 FI ;
143                         OD ;
144                         IF SUCC (col)/=col_2
145                                 THEN RETURN FALSE ;
146                         FI ;
147                 ( TRUE ),( FALSE ): col := col_1;
148                         DO FOR lin := lin_1+1 TO lin_2-1;
149                                 col := PRED (col);
150                                 IF b (lin)(col).status/=free
151                                     THEN RETURN FALSE ;
152                                 FI ;
153                         OD ;
154                         IF PRED (col)/=col_2
155                                 THEN RETURN FALSE ;
156                         FI ;
157                 ( FALSE ),( TRUE ): col := col_1;
158                         DO FOR lin := lin_1-1 DOWN TO lin_2+1;
159                                 col := SUCC (col);
160                                 IF b (lin)(col).status/=free
161                                     THEN RETURN FALSE ;
162                                 FI ;
163                         OD ;
164                         IF SUCC (col)/=col_2
165                                 THEN RETURN FALSE ;
166                         FI ;
167                 ( FALSE ),( FALSE ): col := col_1;
168                         DO FOR lin := lin_1-1 DOWN TO lin_2+1;
169                                 col := PRED (col);
170                                 IF b (lin)(col).status/=free
171                                     THEN RETURN FALSE ;
172                                 FI ;
173                         OD ;
174                         IF PRED (col)/=col_2
175                                 THEN RETURN FALSE ;
176                         FI ;
177             ESAC ;
178             IF arriving.status=free THEN RETURN TRUE ;
179             ELSE RETURN arriving.p.color/=starting.p.color; FI ;
```

```
180         OD ;
181     END ok_bishop;
182     END chess_fragments;
```

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
10           insert: PROC (new node);
11               /* insert actions */
12           END insert;
13
14           remove: PROC ();
15               /* remove actions */
16           END remove;
17
18           initialize_list:
19           BEGIN
20               DCL last REF node := ->head;
21               DO FOR new IN pool;
22                   new.pred := last;
23                   last->.suc := ->new;
24                   last := ->new;
25                   new.value := 0;
26               OD ;
27              head.pred := last;
28              last->.suc := ->head;
29           END initialize_list;
30
31       END handle_list;
32       manipulate:
33        MODULE
34            SEIZE node, remove, insert;
35            DCL node_a node := (: NULL , NULL ,536 :);
36           remove();
37           remove();
38           insert(node_a);
39       END manipulate;
40   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
8        allocate:
9        PROC ()(resource_set);
10           DO FOR EVER ;
11               DO FOR i IN resource_set;
12                   IF NOT allocated(i)
13                       THEN
14                           allocated(i) := TRUE ;
15                            RETURN i;
16                   FI ;
17               OD ;
18               DELAY resource_freed;
19           OD ;
20       END allocate;
21
22       deallocate:
23       PROC (i resource_set);
24           allocated(i) := FALSE ;
25            CONTINUE resource_freed;
26       END deallocate;
27
28   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 calls 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
12      call_distributor:
13       PROCESS ();
14          wait:
15           PROC (x INT );
16              /*some wait action*/
17           END wait;
18           DO FOR EVER ;
19              wait(10 /*seconds*/);
20               CONTINUE operator_is_ready;
21           OD ;
22       END call_distributor;
23
24      call_process:
25       PROCESS ();
26           DELAY CASE
27          (operator_is_ready): /* some actions */ ;
28          (switch_is_closed): DO FOR i IN INT (1:100);
29                                  CONTINUE operator_is_ready;
30                                 /* empty the queue*/
31                              OD ;
32           ESAC ;
33       END call_process;
34
35      operator:
36       PROCESS ();
37           DCL time INT ;
38           DO FOR EVER ;
39              IF time = 1700
40                  THEN CONTINUE switch_is_closed;
41              FI ;
42           OD ;
43       END operator;
44
45       START call_distributor();
46       START operator();
47       DO FOR i IN INT (1:100);
48            START call_process();
49       OD ;
50    END switchboard;
```

15. Allocating and deallocating a set of resources

```
1   <> FREE ( STEP );
2    MODULE
3    SIGNAL
4        acquire,
5        release=( INSTANCE ),
6        congested,
7        ready,
8        step,
9        readout=( INT );
10        GRANT ALL ;
11   END definitions;
12  counter_manager:
13   MODULE
14  /*   To illustrate the use of signals and the receive case, (buffers
15       might have been used instead) we will look at an example where an
16       allocator manages a set of resources, in this case a set of
17       counters. The module is part of a larger system where there are
18       users, that can request the services of the counter_manager. The
19       module is made to consist of two process definitions, one for the
20       allocation and one for the counters. initiate and terminate
21       are internal signals sent from the allocator
22       to the counters. All the other signals are external, being sent
23       from or to the users. */
24
25        SEIZE /* external signals */
26            acquire, release, congested,ready,step,readout;
27        SIGNAL initiate = ( INSTANCE ),
28                terminate;
29       allocator:
30        PROCESS ();
31            NEWMODE no_of_counters = INT (1:100);
32        DCL counters ARRAY (no_of_counters)
33                                STRUCT (counter INSTANCE ,status SET (busy,idle));
34        DO FOR each IN counters;
35            each := (: START counter(), idle :);
36        OD ;
37        DO FOR EVER ;
38        BEGIN
39            DCL user INSTANCE ;
40           await_signals:
41            RECEIVE CASE SET user;
42           (acquire):
43                DO FOR each IN counters;
44                    DO WITH each;
45                        IF status = idle
46                            THEN
47                                status := busy;
48                                SEND initiate (user) TO counter;
49                                EXIT await_signals;
50                        FI ;
51                    OD ;
52                OD ;
53                SEND congested TO user;
54           (release IN this_counter):
55                SEND terminate TO this_counter;
56               find_counter:
57                DO FOR each IN counters;
58                    DO WITH each;
```

```
59                      IF this_counter = counter
60                          THEN
61                              status := idle;
62                               EXIT find_counter;
63                      FI ;
64                  OD ;
65              OD find_counter;
66          ESAC await_signals;
67      END ;
68      OD ;
69      END allocator;
70      counter:
71      PROCESS ();
72          DO FOR EVER ;
73          BEGIN
74              DCL user INSTANCE ,
75                  count INT := 0;
76              RECEIVE CASE
77                  (initiate IN received_user):
78                       SEND ready TO received_user;
79                      user := received_user;
80              ESAC ;
81              work_loop:
82              DO FOR EVER ;
83                  RECEIVE CASE
84                      (step): count + := 1;
85                      (terminate):
86                           SEND readout(count) TO user;
87                           EXIT work_loop;
88                  ESAC ;
89              OD work_loop;
90          END ;
91          OD ;
92      END counter;
93      START allocator();
94  END counter_manager;
```

16. **Allocating and deallocating a set of resources using buffers**

```
1   <> FREE ( STEP );
2
3   user_world:
4    MODULE
5   /* This example is the same as no.15 except that buffers are
6       used for communication in stead of signals.
7       The main difference is that processes are now identified
8       by means of references to local message buffers rather than
9       by instance values. There is one message buffer declared
10      local to each process. There is one set of message types
11      for each process definition. When started each process must
12      identify its buffer address to the starting process.
13      The user_world module sketches some of the environment in
14      which the counter_manager is used. */
15
16  SEIZE allocator;
17  GRANT user_buffers,user_messages,
18            allocator_messages, allocator_buffers,
19            counter_messages, counters_buffers;
20  NEWMODE
21      user_messages =
22          STRUCT (type SET (congested, ready,
23                      readout, allocator_id),
24                                  CASE type OF
25                                      (congested) : ,
26                                      (ready) : counter REF counters_buffers,
27                                      (readout) : count INT ,
28                                      (allocator_id): allocator REF allocator_buffers
29                                  ESAC ),
30      user_buffers = BUFFER (1) user_messages,
31      allocator_messages =
32          STRUCT (type SET (acquire, release, counter_id),
33                      CASE type OF
34                          (acquire) : user REF user_buffers,
35                          (release,
36                          counter_id): counter REF counters_buffers
37                      ESAC ),
38      allocator_buffers = BUFFER (1) allocator_messages,
39      counter_messages =
40          STRUCT (type SET (initiate, step, terminate),
41                      CASE type OF
42                          (initiate) : user REF user_buffers,
43                          (step,
44                          terminate):
45                      ESAC ),
46      counters_buffers = BUFFER (1) counter_messages;
47   DCL user_buffer user_buffers,
48         allocator_buf REF allocator_buffers,
49         counter_buf REF counters_buffers;
50   START allocator(->user_buffer);
51  allocator_buf := ( RECEIVE user_buffer).allocator;
52   END user_world;
53  counter_manager:
54   MODULE
55   SEIZE user_buffers,user_messages,
56         allocator_messages, allocator_buffers,
57         counter_messages, counters_buffers;
58   GRANT allocator;
```

```
59
60  allocator:
61   PROCESS (starter REF user_buffers);
62       DCL allocator_buffer allocator_buffers;
63       NEWMODE no_of_counters = INT (1:10);
64       DCL counters ARRAY (no_of_counters)
65                                  STRUCT (counter REF counters_buffers,
66                                          status SET (busy, idle)),
67            message allocator_messages;
68       SEND starter->([allocator_id, ->allocator_buffer]);
69       DO FOR each IN counters;
70           START counter(->allocator_buffer);
71          each := [( RECEIVE allocator_buffer).counter, idle];
72       OD ;
73       DO FOR EVER ;
74           BEGIN
75           DCL user REF user_buffers;
76          message := RECEIVE allocator_buffer;
77          handle_messages:
78           CASE message.type OF
79          (acquire):
80               user := message.user;
81               DO FOR each IN counters;
82                   DO WITH each;
83                       IF status= idle
84                           THEN status := busy;
85                                   SEND counter->([initiate, user]);
86                                   EXIT handle_messages;
87                       FI ;
88                   OD ;
89               OD ;
90               SEND user->([congested]);
91          (release):
92               SEND message.counter->([terminate]);
93              find_counter:
94               DO FOR each IN counters;
95                   DO WITH each;
96                       IF message.counter = counter
97                           THEN status := idle;
98                                   EXIT find_counter;
99                       FI ;
100                  OD ;
101              OD find_counter;
102         (counter_id): ;
103          ESAC handle_messages;
104          END ;
105      OD ;
106  END allocator;
107 counter:
108  PROCESS (starter REF allocator_buffers);
109      DCL counter_buffer counters_buffers;
110      SEND starter->([counter_id, ->counter_buffer]);
111      DO FOR EVER ;
112          BEGIN
113              DCL user REF user_buffers,
114                  count INT := 0,
115                  message counter_messages;
116              message := RECEIVE counter_buffer;
117              CASE message.type OF
118                  (initiate): user := message.user;
119                              SEND user->([ready, ->counter_buffer]);
```

```
120                 ELSE /* some error action */
121             ESAC ;
122         work_loop:
123          DO FOR EVER ;
124             message := RECEIVE counter_buffer;
125             CASE message.type OF
126                 (step) : count + := 1;
127                 (terminate): SEND user->([readout, count]);
128                              EXIT work_loop;
129                 ELSE /* some error action */
130             ESAC ;
131         OD work_loop;
132     END ;
133     OD ;
134 END counter;
135 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 scanner;
37
38   END string_scanner1;
```

18. **String scanner2**

```
1   string_scanner2: /* This example is the same as no.17 but it uses
2                       character string instead 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 scanner;
38
39   END string_scanner2;
```

19. **Removing an item from a double linked list**

```
1   queue: MODULE
2       SYNMODE info= INT ;
3       queue_removal:
4        MODULE
5           SEIZE info;
6           GRANT remove;
7           remove:
8           PROC (p PTR ) RETURNS (info) EXCEPTIONS (EMPTY);
9                   /* This procedure removes the item referred to
10                     by p from a queue and returns the information
11                     contents of that queue element */
12                  DCL 1 x BASED (p),
13                        2 i info POS (0,8:31),
14                        2 prev PTR POS (1,0:15),
15                        2 next PTR POS (1,16:31);
16                  DCL prev, next PTR ;
17                  prev := x.prev;
18                  next := x.next;
19                  x.prev, x.next := NULL ;
20                  RESULT x.i;
21                  p := prev;
22                  x.next := next;
23                  p := next;
24                  x.prev := prev;
25          END remove;
26      END queue_removal;
27  END queue;
```

20. **Update a record of a file**

```
1   read_modify_write:
2   MODULE
3
4     /* this example indicates how the CHILL i/o concepts can be used */
5     /* to write an application where a record of a random accessible  */
6     /* file can be updated or added if not yet in use                 */
7
8     NEWMODE
9       index_set = INT (1:1000),
10      record_type = STRUCT (
11                                free   BOOL ,
12                                count  INT ,
13                                name   CHAR (20));
14
15    DCL
16      curindex          index_set,
17      file_association   ASSOCIATION ,
18      record_file        ACCESS (index_set) record_type,
19      record_buffer     record_type;
20
21    ASSOCIATE (file_association,'DSK:RECORDS.DAT');     /* create association */
22     CONNECT (record_file,file_association, READWRITE ); /* connect to file    */
23    curindex := 123;                                    /* position record    */
24     READRECORD (record_file,curindex,record_buffer);    /* read the record    */
25       IF record_buffer.free                             /* if record is free  */
26         THEN                                            /* the claim and      */
27           record_buffer.free := FALSE                   /* initialize it      */
28           record_buffer.count := 0;
29           record_buffer.name := 'CHILL I/O concept    ';
30       FI ;
31    record_buffer.count + := 1;                         /* increment its count*/
32     WRITERECORD (record_file, curindex, record_buffer); /* write the record   */
33     DISSOCIATE (file_association);                      /* end the association*/
34
35  END read_modify_write;
```

```
1   merge_sorted_files:
2   MODULE
3
4     /* this example shows how two sorted files can be merged into one */
5     /* new sorted file, where the field 'key' is used for sorting      */
6     /* the old sorted files are deleted after the merging has been done */
7
8     NEWMODE
9       record_type = STRUCT (
10                                key    INT ,
11                                name   CHAR (50));
12
13    DCL
14      flag        BOOL ,
15      infiles     ARRAY ( BOOL ) ACCESS record_type,
16      outfile     ACCESS record_type,
17      buffers     ARRAY ( BOOL ) record_type,
18      innames     ARRAY ( BOOL ) CHAR (10) INIT := ['FILE.IN.1 ','FILE.IN.2 '],
19      outname     CHAR (10) INIT := 'FILE.OUT   ',
20      inassocs    ARRAY ( BOOL ) ASSOCIATION ,
21      outassoc    ASSOCIATION ;
22
23    /* associate both sorted input files, connect an access to them for input */
24    /* and read their first record into a buffer                              */
25
26    DO
27      FOR curfile IN infiles,
28          curbuffer IN buffers,
29          curassoc IN inassocs,
30          curname IN innames;
31        CONNECT (curfile, ASSOCIATE (curassoc,curname), READONLY );
32        READRECORD (curfile, curbuffer);
33    OD ;
34
35    /* associate the output file, create a file for the association */
36    /* and connect an access to it for output                       */
37
38    ASSOCIATE (outassoc,outname);
39    CREATE (outassoc);
40    CONNECT (outfile, outassoc, WRITEONLY );
41    merge_files:
42      DO FOR EVER
43
44        /* determine which file, if any at all, to process next*/
45        /* 'flag' indicates the file                           */
46
47        CASE OUTOFFILE (infiles( FALSE )), OUTOFFILE (infiles( TRUE )) OF
48          ( TRUE ),  ( TRUE ):                    /* both files are empty */
49                      EXIT merge_files;
50          ( TRUE ),  ( FALSE ):                   /* one file is empty    */
51                     flag := TRUE ;
52          ( FALSE ), ( TRUE ):                    /* one file is empty    */
53                     flag := FALSE ;
54          ( FALSE ), ( FALSE ):                   /* no file is empty     */
55                     flag := buffers( FALSE ).key>buffers( TRUE ).key;
56        ESAC ;
57
58          /* output the buffer which currently contains a record with the */
```

```
59              /* smallest value for 'key', fill the buffer with a new record      */
60
61              WRITERECORD (outfile,buffers(flag));
62              READRECORD (infiles(flag), buffers(flag));
63      OD merge_files;
64
65      /* delete the input files and close the output file */
66
67      DO
68        FOR curassoc IN inassocs;
69          DELETE (curassoc);                    /* delete the file              */
70          DISSOCIATE (curassoc);                /* and terminate association    */
71      OD ;
72       DISSOCIATE (outassoc);                   /* disconnect and terminate     */
73
74  END merge_sorted_files;
```

22. Read a file with variable length records

```
1   variable_length_records:
2    MODULE
3
4      /* This example shows how a file which consists of variable length   */
5      /* records can be treated.                                           */
6      /* The file consists of a number of strings of varying length; the   */
7      /* algorithm will read a string, allocate an appropriate location    */
8      /* for it, and put the reference to this location into a push down list */
9
10     NEWMODE
11       string = CHAR (80),
12       link_record = STRUCT (
13                                 next_record    REF link_record,
14                                 string_row     ROW string);
15
16     DCL
17       pushdownlist    REF link_record INIT := NULL ,
18       length          INT (1:80),
19       temporaryrow    ROW string,
20       fileaccess      ACCESS string DYNAMIC ,
21       association     ASSOCIATION ;
22
23     ASSOCIATE (association,'INPUT.DATA');          /* associate the input file     */
24     CONNECT (fileaccess, association, READONLY );  /* connect access for input     */
25    temporaryrow := READRECORD (fileaccess);        /* read the first record        */
26     DO                                             /* while not end-of-file        */
27       WHILE NOT(OUTOFFILE(fileaccess));
28         pushdownlist := ALLOCATE (link_record,     /* get a new link record        */
29                                   [pushdownlist, NULL ]);  /* and initialize it      */
30         length := 1 + UPPER (temporaryrow->);      /* determine length of string   */
31          DO
32            WITH pushdowlist->;                     /* add new string to list       */
33             string_row := ALLOCATE ( CHAR (length), /* allocate space for string   */
34                                      temporaryrow->);  /* and fill it               */
35          OD ;
36         temporaryrow := READRECORD (fileaccess);   /* get next record in file      */
37     OD ;
38     DISSOCIATE (association);                      /* end the association          */
39
40   END variable_length_records;
```

23. **The use of spec modules**

```
1   letter_count:
2   SPEC MODULE
3       /* This is a spec module for the corresponding module in example 8. */
4       SEIZE max;
5       count: PROC ( ROW CHAR (max) IN , ARRAY ('A':'Z') INT OUT ); END ;
6       GRANT count;
7   END letter_count;
8   test:
9   MODULE
10      /* This is the module 'test' from example 8.          */
11      /* It can now be piecewise compiled together with     */
12      /* the above spec module                              */
13      SYNMODE results = ARRAY ('A':'Z') INT ;
14      DCL c CHAR (10) INIT := 'A-B>ZAA9K' ' ';
15      SYN max = 10_000;
16      GRANT max;
17      SEIZE count;
18      count (-> c, output);
19      ASSERT output = results [('A') : 3, ('B', 'K', 'Z') : 1, ( ELSE ) : 0 ];
20  END test;
```

24. **Example of a context**

```
1   CONTEXT
2       /* This is a context for the same module "test"   */
3       /* as used in example 23, allowing the piecewise  */
4       /* compilation of "test"                          */
5       count : PROC ( ROW CHAR (max) IN , ARRAY ('A':'Z') INT OUT ); END ;
6   END FOR
7
8   test :
9   MODULE
10      SYNMODE results = ARRAY ('A':'Z') INT ;
11      DCL c CHAR (10) INIT := 'A-B>ZAA9K' ' ';
12      SYN max = 10_000;
13      GRANT max;
14      SEIZE count;
15      count (-> c, output);
16      ASSERT output = results [('A') : 3, ('B', 'K', 'Z') : 1, ( ELSE ) : 0 ];
17  END test;
```

25. The use of prefixing and remote modules

```
1     /* This example uses the module 'stacks_1' from example 10.   */
2     /* It shows how prefixes can be used to prevent name clashes. */
3     /* It uses the remote construct to share source code.         */
4     /* It is assumed that the code of the module 'stacks_1' can   */
5     /* be referred to through the text reference name 'stack_code' */
6  char_stack:
7  MODULE
8     SYNMODE element = CHAR ;
9     MODULE REMOTE stack_code ;
10    GRANT ALL PREFIXED stack ! char ;
11 END char_stack ;
12
13 int_stack:
14 MODULE
15    SYNMODE element = INT ;
16    MODULE REMOTE stack_code ;
17    GRANT ALL PREFIXED stack ! int ;
18 END int_stack ;
19    /* Here 'push', 'pop' and 'element' are visible but           */
20    /* with prefixes 'stack ! char' and 'stack ! int' for         */
21    /* the implementations with element = CHAR and                */
22    /* element = INT respectively.                                */
23    /* Below are some possibilities of using the granted          */
24    /* names inside modules.                                      */
25 MODULE
26    SEIZE ALL PREFIXED stack ;
27    DCL c CHAR ;
28    int ! push (123) ;
29    char ! push ('a') ;
30    int ! pop ( ) ;
31    c = char ! elem (1) ;
32 END ;
33
34 MODULE
35    SEIZE (stack ! int -> stack) ! ALL ;
36    stack ! push (345) ;
37    stack ! pop ( ) ;
38 END ;
```

APPENDIX E: COLLECTED SYNTAX

2 PRELIMINARIES

2.2 VOCABULARY

<simple name string> ::=
    <letter> { *<letter>* | *<digit>* | _}*

2.4 COMMENTS

<comment> ::=
    /* *<character string>* */

<character string> ::=
    { *<character>* } *

2.6 COMPILER DIRECTIVES

<directive clause> ::=
    <> *<directive>*{,*<directive>*}*[<>]

<directive> ::=
    <CHILL directive>
  | *<implementation directive>*

<CHILL directive> ::=
    <free directive>

<free directive> ::=
    FREE (*<reserved simple name string list>*)

<simple name string list> ::=
    <simple name string>{,*<simple name string>*} *

2.7 NAMES AND THEIR DEFINING OCCURRENCES

<name> ::=
    <name string>

<name string> ::=
    <simple name string>
  | *<prefixed name string>*

<prefixed name string> ::=
    <prefix> ! *<simple name string>*

<prefix> ::=
    <simple prefix> { ! *<simple prefix>* } *

<simple prefix> ::=
    <simple name string>

<defining occurrence> ::=
    <simple name string>

<defining occurrence list> ::=
    <defining occurrence>{, *<defining occurrence>* } *

<*field name*> ::=
        <*simple name string*>

<*field name defining occurrence*> ::=
        <*simple name string*>

<*field name defining occurrence list*> ::=
        <*field name defining occurrence*> {, <*field name defining occurrence*> } *

<*exception name*> ::=
        <*simple name string*>
      | <*prefixed name string*>

<*register name*> ::=
        <*simple name string*>
      | <*prefixed name string*>

<*text reference name*> ::=
        <*simple name string*>
      | <*prefixed name string*>

<*map reference name*> ::=
        <*simple name string*>
      | <*prefixed name string*>

3 MODES AND CLASSES

3.2 MODE DEFINITIONS

3.2.1 General

<*mode definition*> ::=
        <*defining occurrence list*> = <*defining mode*>

<*defining mode*> ::=
        <*mode*>

3.2.2 Synmode definitions

<*synmode definition statement*> ::=
        SYNMODE <*mode definition*> { , <*mode definition*>}*;

3.2.3 Newmode definitions

<*newmode definition statement*> ::=
        NEWMODE <*mode definition*> { , <*mode definition*>} *;

3.3 MODE CLASSIFICATION

<*mode*> ::=
        [**READ**] <*non-composite mode*>
      | [**READ**] <*composite mode*>

<*non-composite mode*> ::=
        <*discrete mode*>
      | <*powerset mode*>
      | <*reference mode*>
      | <*procedure mode*>

| <*instance mode*>
| <*synchronisation mode*>
| <*input-output mode*>

3.4 DISCRETE MODES

3.4.1 General

<*discrete mode*> ::=
<*integer mode*>
| <*boolean mode*>
| <*character mode*>
| <*set mode*>
| <*range mode*>

3.4.2 Integer modes

<*integer mode*> ::=
INT
| *BIN*
| <*integer mode* name>

3.4.3 Boolean modes

<*boolean mode*> ::=
BOOL
| <*boolean mode* name>

3.4.4 Character modes

<*character mode*> ::=
CHAR
| <*character mode* name>

3.4.5 Set modes

<*set mode*> ::=
SET (<*set list*>)
| <*set mode* name>

<*set list*> ::=
<*numbered set list*>
| <*unnumbered set list*>

<*numbered set list*> ::=
<*numbered set element*> { ,<*numbered set element*>}*

<*numbered set element*> ::=
<*defining occurrence*> = <*integer literal* expression>

<*unnumbered set list*> ::=
<*set element*> { ,<*set element*>} *

<*set element*> ::=
<*defining occurrence*>
| <*unnamed value*>

<*unnamed value*> ::=

*

3.4.6 Range modes

<*range mode*> ::=
<*discrete mode* name>(<*literal range*>)
| **RANGE** (<*literal range*>)
| *BIN* (<*integer literal* expression>)
| <*range mode* name>

<*literal range*> ::=
<*lower bound*> : <*upper bound*>

<*lower bound*> ::=
<*discrete literal* expression>

<*upper bound*> ::=
<*discrete literal* expression>

3.5 POWERSET MODES

<*powerset mode*> ::=
POWERSET <*member mode*>
| <*powerset mode* name>

<*member mode*> ::=
<*discrete* mode>

3.6 REFERENCE MODES

3.6.1 General

<*reference mode*> ::=
<*bound reference mode*>
| <*free reference mode*>
| <*row mode*>

3.6.2 Bound reference modes

<*bound reference mode*> ::=
REF <*referenced mode*>
| <*bound reference mode* name>

<*referenced mode*> ::=
<*mode*>

3.6.3 Free reference modes

<*free reference mode*> ::=
PTR
| <*free reference mode* name>

3.6.4 Row modes

<*row mode*> ::=
ROW <*string* mode>
| **ROW** <*array* mode>
| **ROW** <*variant structure mode* name>

| <row mode name>

3.7 PROCEDURE MODES

<procedure mode> ::=
PROC ([<parameter list>]) [<result spec>]
[**EXCEPTIONS** (<exception list>)] [**RECURSIVE**]
| <procedure mode name>

<parameter list> ::=
<parameter spec> { ,<parameter spec>} *

<parameter spec> ::=
<mode> [<parameter attribute>] [<register name>]

<parameter attribute> ::=
IN | **OUT** | **INOUT** | **LOC** [**DYNAMIC**]

<result spec> ::=
[**RETURNS**] (<mode> [<result attribute>] [<register name>])

<result attribute>::=
[**NONREF**] **LOC** [**DYNAMIC**]

<exception list> ::=
<exception name> { ,<exception name>} *

3.8 INSTANCE MODES

<instance mode> ::=
INSTANCE
| <instance mode name>

3.9 SYNCHRONISATION MODES

3.9.1 General

<synchronisation mode> ::=
<event mode>
| <buffer mode>

3.9.2 Event modes

<event mode> ::=
EVENT [(<event length>)]
| <event mode name>

<event length> ::=
<integer literal expression>

3.9.3 Buffer modes

<buffer mode> ::=
BUFFER [(<buffer length>)]<buffer element mode>
| <buffer mode name>

<buffer length> ::=
<integer literal expression>

<*buffer element mode*> ::=
<*mode*>

3.10 INPUT-OUTPUT MODES

3.10.1 General

<*input-output mode*> ::=
<*association mode*>
| <*access mode*>

3.10.2 Association modes

<*association mode*> ::=
ASSOCIATION
| <*association mode name*>

3.10.3 Access modes

<*access mode*> ::=
ACCESS [(<*index mode*>)] [<*record mode*> [**DYNAMIC**]]
| <*access mode name*>

<*record mode*> ::=
<*mode*>

<*index mode*> ::=
<*discrete mode*>
| <*literal range*>

3.11 COMPOSITE MODES

3.11.1 General

<*composite mode*> ::=
<*string mode*>
| <*array mode*>
| <*structure mode*>

3.11.2 String modes

<*string mode*> ::=
<*string type*>(<*string length*>)
| <*parameterised string mode*>
| <*string mode name*>

<*parameterised string mode*> ::=
<*origin string mode name*>(<*string length*>)
| <*parameterised string mode name*>

<*origin string mode name*> ::=
<*string mode name*>

<*string type*> ::=
CHAR
| *BIT*

<*string length*> ::=

<integer literal expression>

3.11.3 Array modes

<array mode> ::=
[**ARRAY**] (<index mode> { ,<index mode>}*)
<element mode> { <element layout>} *
| <parameterised array mode>
| <array mode name>

<parameterised array mode> ::=
<origin array mode name>(<upper index>)
| <parameterised array mode name>

<origin array mode name> ::=
<array mode name>

<upper index> ::=
<discrete literal expression>

<element mode> ::=
<mode>

3.11.4 Structure modes

<structure mode> ::=
<nested structure mode>
| <level structure mode>
| <parameterised structure mode>
| <structure mode name>

<nested structure mode> ::=
STRUCT (<fields> { ,<fields>} *)

<fields> ::=
<fixed fields>
| <alternative fields>

<fixed fields> ::=
<field name defining occurrence list> <mode> [<field layout>]

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

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

<tags> ::=
<tag field name> { ,<tag field name>} *

<variant fields> ::=
<field name defining occurrence list> <mode> [<field layout>]

<parameterised structure mode> ::=
<origin variant structure mode name> (<literal expression list>)
| <parameterised structure mode name>

<origin variant structure mode name> ::=
<variant structure mode name>

<literal expression list> ::=
<discrete literal expression> { ,*<discrete literal expression>*} *

3.11.5 Level structure notation

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

<(n) level fields> ::=
<(n) level fixed fields>
| *<(n) level alternative fields>*

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

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

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

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

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

3.11.6 Layout description for array modes and structure modes

<element layout> ::=
PACK | **NOPACK** | *<step>*

<field layout> ::=
PACK | **NOPACK** | *<pos>*

<step> ::=
STEP (*<pos>* [,*<step size>*])

<pos> ::=
POS (*<word>* ,*<start bit>* ,*<length>*)
| **POS** (*<word>* [,*<start bit>* [: *<end bit>*]])

<word> ::=
<integer literal expression>
| *<map reference name>*

<step size> ::=
<integer literal expression>

<*start bit*> ::=
        <*integer literal expression*>

<*end bit*> ::=
        <*integer literal expression*>

<*length*> ::=
        <*integer literal expression*>

4 LOCATIONS AND THEIR ACCESSES

4.1 DECLARATIONS

4.1.1 General

<*declaration statement*> ::=
        DCL <*declaration*> { ,<*declaration*>} *;

<*declaration*> ::=
        <*location declaration*>
      | <*loc-identity declaration*>
      | <*based declaration*>

4.1.2 Location declarations

<*location declaration*> ::=
        <*defining occurrence list*> <*mode*> [**STATIC**] [<*initialisation*>]

<*initialisation*> ::=
        <*reach-bound initialisation*>
      | <*lifetime-bound initialisation*>

<*reach-bound initialisation*> ::=
        <*assignment symbol*> <*value*> [<*handler*>]

<*lifetime-bound initialisation*> ::=
        INIT <*assignment symbol*> <*constant value*>

4.1.3 Loc-identity declarations

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

4.1.4 Based declarations

<*based declaration*> ::=
        <*defining occurrence list*> <*mode*> **BASED**
        [(<*bound or free reference location name*>)]

4.2 LOCATIONS

4.2.1 General

<*location*> ::=
        <*access name*>

| <*dereferenced bound reference*>
| <*dereferenced free reference*>
| <*dereferenced row*>
| <*string element*>
| <*string slice*>
| <*array element*>
| <*array slice*>
| <*structure field*>
| <*location procedure call*>
| <*location built-in routine call*>
| <*location conversion*>

4.2.2 Access names

<*access name*> ::=
<*location* name>
| <*loc-identity* name>
| <*based* name>
| <*location enumeration* name>
| <*location do-with* name>

4.2.3 Dereferenced bound references

<*dereferenced bound reference*> ::=
<*bound reference primitive value*> -> [<*mode* name>]

4.2.4 Dereferenced free references

<*dereferenced free reference*> ::=
<*free reference primitive value*> -> <*mode* name>

4.2.5 Dereferenced rows

<*dereferenced row*> ::=
<*row primitive value*> ->

4.2.6 String elements

<*string element*> ::=
<*string location*> (<*start element*>)

4.2.7 String slices

<*string slice*> ::=
<*string location*> (<*left element*> : <*right element*>)
| <*string location*> (<*start element*> **UP** <*slice size*>)

<*left element*> ::=
<*integer expression*>

<*right element*> ::=
<*integer expression*>

<*start element*> ::=
<*integer expression*>

<*slice size*> ::=
<*integer expression*>

4.2.8 Array elements

<array element> ::=
<array location> (<expression list>)

<expression list> ::=
*<expression> { , <expression>} **

4.2.9 Array slices

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

<lower element> ::=
<expression>

<upper element> ::=
<expression>

<first element> ::=
<expression>

4.2.10 Structure fields

<structure field> ::=
<structure location> . <field name>

4.2.11 Location procedure calls

<location procedure call> ::=
<location procedure call>

4.2.12 Location built-in routine calls

<location built-in routine call> ::=
<implementation location built-in routine call>
| <CHILL location built-in routine call>

<CHILL location built-in routine call> ::=
<io CHILL location built-in routine call>

4.2.13 Location conversions

<location conversion> ::=
<mode name>(<static mode location>)

5 VALUES AND THEIR OPERATIONS

5.1 SYNONYM DEFINITIONS

<synonym definition statement> ::=
SYN *<synonym definition> { ,<synonym definition>} *;*

<synonym definition> ::=
<defining occurrence list> [<mode>] = <constant value>

5.2 PRIMITIVE VALUE

5.2.1 General

<primitive value> ::=
    <location contents>
  | *<value name>*
  | *<literal>*
  | *<tuple>*
  | *<value string element>*
  | *<value string slice>*
  | *<value array element>*
  | *<value array slice>*
  | *<value structure field>*
  | *<expression conversion>*
  | *<value procedure call>*
  | *<value built-in routine call>*
  | *<start expression>*
  | *<zero-adic operator>*
  | *<parenthesised expression>*

5.2.2 Location contents

<location contents> ::=
    <location>

5.2.3 Value names

<value name> ::=
    <synonym name>
  | *<value enumeration name>*
  | *<value do-with name>*
  | *<value receive name>*
  | *<general procedure name>*

5.2.4 Literals

5.2.4.1 General

<literal> ::=
    <integer literal>
  | *<boolean literal>*
  | *<set literal>*
  | *<emptiness literal>*
  | *<character string literal>*
  | *<bit string literal>*

5.2.4.2 Integer literals

<integer literal> ::=
    <decimal integer literal>
  | *<binary integer literal>*
  | *<octal integer literal>*
  | *<hexadecimal integer literal>*

<decimal integer literal> ::=
    [*D'*] { *<digit>* | _ } $^{+}$

<*binary integer literal*> ::=
B' { *0* | *1* | _} +

<*octal integer literal*> ::=
O' { <*octal digit*> | _} +

<*hexadecimal integer literal*> ::=
H' { <*hexadecimal digit*> | _} +

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

<*hexadecimal digit*> ::=
<*digit*> | *A* | *B* | *C* | *D* | *E* | *F*

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

5.2.4.3 Boolean literals

<*boolean literal*> ::=
FALSE | *TRUE*

5.2.4.4 Set literals

<*set literal*> ::=
<*set element* name>

5.2.4.5 Emptiness literal

<*emptiness literal*> ::=
NULL

5.2.4.6 Character string literals

<*character string literal*> ::=
' { <*non-apostrophe character*> | <*apostrophe*>}*'
| *C'* { <*octal digit*> <*hexadecimal digit*> | _ }*'

<*character*> ::=
<*letter*>
| <*digit*>
| <*symbol*>
| <*space*>

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

<*symbol*> ::=
" | ' | (|) | * | + | , | - | . | / | : | ; | < | = | > | ?

<*space*> ::=
SP

<*apostrophe*> ::=
' '

5.2.4.7 Bit string literals

<bit string literal> ::=
<binary bit string literal>
| *<octal bit string literal>*
| *<hexadecimal bit string literal>*

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

<octal bit string literal> ::=
O' { *<octal digit>* | *_* } * *'*

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

5.2.5 Tuples

<tuple> ::=
[*<mode name>*] *(:* { *<powerset tuple>* | *<array tuple>* | *<structure tuple>* } *:)*
| *<character string literal>*
| *<bit string literal>*

<powerset tuple> ::=
[{ *<expression>* | *<range>* } { , { *<expression>* | *<range>* } } *]

<range> ::=
<expression> : *<expression>*

<array tuple> ::=
<unlabelled array tuple>
| *<labelled array tuple>*

<unlabelled array tuple> ::=
<value> { ,*<value>* } *

<labelled array tuple> ::=
<case label list> : *<value>* { , *<case label list>* : *<value>* } *

<structure tuple> ::=
<unlabelled structure tuple>
| *<labelled structure tuple>*

<unlabelled structure tuple> ::=
<value> { ,*<value>* } *

<labelled structure tuple> ::=
<field name list> : *<value>* { , *<field name list>* : *<value>* } *

<field name list> ::=
.*<field name>* { , .*<field name>* } *

5.2.6 Value string elements

<value string element> ::=
<string primitive value>(<start element>)

5.2.7 Value string slices

<value string slice> ::=
<string primitive value>(<left element> : <right element>)
| *<string primitive value>(<start element>* **UP** *<slice size>)*

5.2.8 Value array elements

<value array element> ::=
    <array primitive value> (<expression list>)

5.2.9 Value array slices

<value array slice> ::=
    <array primitive value> (<lower element> : <upper element>)
  | <array primitive value> (<first element> **UP** <slice size>)

5.2.10 Value structure fields

<value structure field> ::=
    <structure primitive value> . <field name>

5.2.11 Expression conversions

<expression conversion> ::=
    <mode name> (<expression>)

5.2.12 Value procedure calls

<value procedure call> ::=
    <value procedure call>

5.2.13 Value built-in routine calls

<value built-in routine call> ::=
    <implementation value built-in routine call>
  | <CHILL value built-in routine call>

<CHILL value built-in routine call> ::=
    NUM (<discrete expression>)
  | PRED (<discrete expression>)
  | SUCC (<discrete expression>)
  | ABS (<integer expression>)
  | CARD (<powerset expression>)
  | MAX (<powerset expression>)
  | MIN (<powerset expression>)
  | SIZE ({ <mode name> | <static mode location>})
  | UPPER (<upper lower argument>)
  | LOWER (<upper lower argument>)
  | GETSTACK (<getstack argument> [, <value>])
  | ALLOCATE (<allocate argument> [, <value>])
  | <io CHILL value built-in routine call>

<getstack argument> ::=
    <argument>

<allocate argument> ::=
    <argument>

<argument> ::=
    <mode name>
  | <array mode name>(<expression>)
  | <string mode name>(<integer expression>)
  | <variant structure mode name>(<expression list>)

<*upper lower argument*> ::=
<*array* location>
| <*array* primitive value>
| <*array mode* name>
| <*string* location>
| <*string* primitive value>
| <*string mode* name>
| <*discrete* location>
| <*discrete* expression>
| <*discrete mode* name>

5.2.14 Start expressions

<*start expression*> ::=
START <*process* name> ([<*actual parameter list*>])

5.2.15 Zero-adic operator

<*zero-adic operator*> ::=
THIS

5.2.16 Parenthesised expression

<*parenthesised expression*> ::=
(<*expression*>)

5.3 VALUES AND EXPRESSIONS

5.3.1 General

<*value*> ::=
<*expression*>
| <*undefined value*>

<*undefined value*> ::=
*
| <*undefined synonym* name>

5.3.2 Expressions

<*expression*> ::=
<*operand-1*>
| <*sub expression*> { *OR* | *XOR* } <*operand-1*>

<*sub expression*> ::=
<*expression*>

5.3.3 Operand-1

<*operand-1*> ::=
<*operand-2*>
| <*sub operand-1*> *AND* <*operand-2*>

<*sub operand-1*> ::=
<*operand-1*>

5.3.4 Operand-2

<*operand-2*> ::=
        <*operand-3*>
      | <*sub operand-2*> <*operator-3*> <*operand-3*>

<*sub operand-2*> ::=
        <*operand-2*>

<*operator-3*> ::=
        <*relational operator*>
      | <*membership operator*>
      | <*powerset inclusion operator.*

<*relational operator*> ::=
        = | /= | > | >= | < | <=

<*membership operator*> ::=
        IN

<*powerset inclusion operator*> ::=
        <= | >= | < | >

5.3.5 Operand-3

<*operand-3*> ::=
        <*operand-4*>
      | <*sub operand-3*> <*operator-4*> <*operand-4*>

<*sub operand-3*> ::=
        <*operand-3*>

<*operator-4*> ::=
        <*arithmetic additive operator*>
      | <*string concatenation operator*>
      | <*powerset difference operator*>

<*arithmetic additive operator*> ::=
        + | -

<*string concatenation operator*> ::=
        //

<*powerset difference operator*> ::=
        -

5.3.6 Operand-4

<*operand-4*> ::=
        <*operand-5*>
      | <*sub operand-4*> <*arithmetic multiplicative operator*> <*operand-5*>

<*sub operand-4*> ::=
        <*operand-4*>

<*arithmetic multiplicative operator*> ::=
        * | / | *MOD* | *REM*

5.3.7 Operand-5

<*operand-5*> ::=
        [<*monadic operator*>] <*operand-6*>

<*monadic operator*> ::=
- | *NOT*
| <*string repetition operator*>

<*string repetition operator*> ::=
(<*integer literal* expression>)

5.3.8 Operand-6

<*operand-6*> ::=
<*referenced location*>
| <*receive expression*>
| <*primitive value*>

<*referenced location*> ::=
-> <*location*>
| **ADDR** (<*location*>)

<*receive expression*> ::=
RECEIVE <*buffer* location>

6 ACTIONS

6.1 GENERAL

<*action statement*> ::=
[<*defining occurrence*> :] <*action*> [<*handler*>] [<*simple name string*>];
| <*module*>
| <*spec module*>

<*action*> ::=
<*bracketed action*>
| <*assignment action*>
| <*call action*>
| <*exit action*>
| <*return action*>
| <*result action*>
| <*goto action*>
| <*assert action*>
| <*empty action*>
| <*start action*>
| <*stop action*>
| <*delay action*>
| <*continue action*>
| <*send action*>
| <*cause action*>

<*bracketed action*> ::=
<*if action*>
| <*case action*>
| <*do action*>
| <*begin-end block*>
| <*delay case action*>
| <*receive case action*>

6.2 ASSIGNMENT ACTION

<*assignment action*> ::=

<*single assignment action*>
| <*multiple assignment action*>

<*single assignment action*> ::=
<*location*> { <*assignment symbol*> | <*assigning operator*>} <*value*>

<*multiple assignment action*> ::=
<*location*> { ,<*location*>}$^{+}$ <*assignment symbol*> <*value*>

<*assigning operator*> ::=
<*closed dyadic operator*> <*assignment symbol*>

<*closed dyadic operator*> ::=
OR | *XOR*
| *AND*
| <*powerset difference operator*>
| <*arithmetic additive operator*>
| <*arithmetic multiplicative operator*>

<*assignment symbol*> ::=
:= | =

6.3 IF ACTION

<*if action*> ::=
IF <*boolean expression*> <*then clause*> [<*else clause*>] **FI**

<*then clause*> ::=
THEN <*action statement list*>

<*else clause*> ::=
ELSE <*action statement list*>
| **ELSIF** <*boolean expression*> <*then clause*> [<*else clause*>]

6.4 CASE ACTION

<*case action*> ::=
CASE <*case selector list*> **OF** [<*range list*>;] { <*case alternative*>} $^{+}$
[**ELSE** <*action statement list*>]
ESAC

<*case selector list*> ::=
<*discrete expression*> { ,<*discrete expression*>} *

<*range list*> ::=
<*discrete mode*> { ,<*discrete mode*>} *

<*case alternative*> ::=
<*case label specification*> : <*action statement list*>

6.5 DO ACTION

6.5.1 General

<*do action*> ::=
DO [<*control part*>;] <*action statement list*> **OD**

<*control part*> ::=
<*for control*> [<*while control*>]
| <*while control*>
| <*with part*>

6.5.2 For control

<for control> ::=
FOR { *<iteration>* { ,*<iteration>*} * | **EVER** }

<iteration> ::=
<value enumeration>
| *<location enumeration>*

<value enumeration> ::=
<step enumeration>
| *<range enumeration>*
| *<powerset enumeration>*

<step enumeration> ::=
<loop counter> *<assignment symbol>*
<start value> [*<step value>*] [**DOWN**] *<end value>*

<loop counter> ::=
<defining occurrence>

<start value> ::=
<discrete expression>

<step value> ::=
BY *<integer expression>*

<end value> ::=
TO *<discrete expression>*

<range enumeration> ::=
<loop counter> [**DOWN**] **IN** *<discrete mode>*

<powerset enumeration> ::=
<loop counter> [**DOWN**] **IN** *<powerset expression>*

<location enumeration> ::=
<loop counter> [**DOWN**] **IN** *<composite location>*

<composite location> ::=
<array location>
| *<string location>*

6.5.3 While control

<while control> ::=
WHILE *<boolean expression>*

6.5.4 With part

<with part> ::=
WITH *<with control>* { ,*<with control>*} *

<with control> ::=
<structure location>
| *<structure primitive value>*

6.6 EXIT ACTION

<exit action> ::=
EXIT *<simple name string>*

6.7 CALL ACTION

<*call action*> ::=
[**CALL**] { <*procedure call*>
| <*CHILL built-in routine call*>
| <*implementation built-in routine call*>}

<*procedure call*> ::=
{ <*procedure name*> | <*procedure primitive value*>} ([<*actual parameter list*>])

<*actual parameter list*> ::=
<*actual parameter*> { ,<*actual parameter*>} *

<*actual parameter*> ::=
<*value*>
| <*location*>

<*CHILL built-in routine call*> ::=
<*CHILL value built-in routine call*>
| <*CHILL location built-in routine call*>
| <*CHILL simple built-in routine call*>

<*CHILL simple built-in routine call*> ::=
TERMINATE (<*reference expression*>)
| <*io CHILL simple built-in routine call*>

6.8 RESULT AND RETURN ACTION

<*return action*> ::=
RETURN [<*result*>]

<*result action*> ::=
RESULT <*result*>

<*result*> ::=
<*value*>
| <*location*>

6.9 GOTO ACTION

<*goto action*> ::=
GOTO <*simple name string*>

6.10 ASSERT ACTION

<*assert action*> ::=
ASSERT <*boolean expression*>

6.11 EMPTY ACTION

<*empty action*> ::=
<*empty*>

<*empty*> ::=

6.12 CAUSE ACTION

<*cause action*> ::=
CAUSE <*exception name*>

6.13 START ACTION

<start action> ::=
<start expression> [**SET** *<instance location>*]

6.14 STOP ACTION

<stop action> ::=
STOP

6.15 CONTINUE ACTION

<continue action> ::=
CONTINUE *<event location>*

6.16 DELAY ACTION

<delay action> ::=
DELAY *<event location>* [*<priority>*]

<priority> ::=
PRIORITY *<integer literal expression>*

6.17 DELAY CASE ACTION

<delay case action> ::=
DELAY CASE [{ **SET** *<instance location>* [*<priority>*] ; | *<priority>*;}]
{ *<delay alternative>*} +
ESAC

<delay alternative> ::=
(*<event list>*) : *<action statement list>*

<event list> ::=
<event location> { ,*<event location>*} *

6.18 SEND ACTION

6.18.1 General

<send action> ::=
<send signal action>
| *<send buffer action>*

6.18.2 Send signal action

<send signal action> ::=
SEND *<signal name>* [(*<value>* { ,*<value>*} *)]
[**TO** *<instance primitive value>*] [*<priority>*]

6.18.3 Send buffer action

<send buffer action> ::=
SEND *<buffer location>*(*<value>*) [*<priority>*]

6.19 RECEIVE CASE ACTION

6.19.1 General

<receive case action> ::=
<receive signal case action>
| *<receive buffer case action>*

6.19.2 Receive signal case action

<receive signal case action> ::=
RECEIVE CASE [**SET** *<instance location>*;]
{ *<signal receive alternative>*} $^{+}$
[**ELSE** *<action statement list>*] **ESAC**

<signal receive alternative> ::=
(*<signal name>* [**IN** *<defining occurrence list>*]) : *<action statement list>*

6.19.3 Receive buffer case action

<receive buffer case action> ::=
RECEIVE CASE [**SET** *<instance location>*;]
{ *<buffer receive alternative>*} $^{+}$
[**ELSE** *<action statement list>*]
ESAC

<buffer receive alternative> ::=
(*<buffer location>* **IN** *<defining occurrence>*) : *<action statement list>*

7 INPUT AND OUTPUT

7.4 BUILT-IN ROUTINES FOR INPUT OUTPUT

7.4.1 General

<io CHILL value built-in routine call> ::=
<association attr io CHILL value built-in routine call>
| *<isassociated io CHILL value built-in routine call>*
| *<access attr io CHILL value built-in routine call>*
| *<readrecord io CHILL value built-in routine call>*

<io CHILL simple built-in routine call> ::=
<dissociate io CHILL simple built-in routine call>
| *<modification io CHILL simple built-in routine call>*
| *<connect io CHILL simple built-in routine call>*
| *<disconnect io CHILL simple built-in routine call>*
| *<writerecord io CHILL simple built-in routine call>*

<io CHILL location built-in routine call> ::=
<associate io CHILL location built-in routine call>

7.4.2 Associating an outside world object

<associate io CHILL location built-in routine call> ::=

ASSOCIATE (<association location>[,<associate parameter list>])

<isassociated io CHILL value built-in routine call> ::=
ISASSOCIATED (<association location>)

<associate parameter list> ::=
*<associate parameter> { ,<associate parameter> } **

<associate parameter> ::=
<location>
| *<value>*

7.4.3 Dissociating an outside world object

<dissociate io CHILL simple built-in routine call> ::=
DISSOCIATE (<association location>)

7.4.4 Accessing association attributes

<association attr io CHILL value built-in routine call> ::=
EXISTING (<association location>)
| *READABLE (<association location>)*
| *WRITEABLE (<association location>)*
| *INDEXABLE (<association location>)*
| *SEQUENCIBLE (<association location>)*
| *VARYING (<association location>)*

7.4.5 Modifying association attributes

<modification io CHILL simple built-in routine call> ::=
CREATE (<association location>)
| *DELETE (<association location>)*
| *MODIFY (<association location>[,<modify parameter list>])*

<modify parameter list> ::=
*<modify parameter> { , <modify parameter> } **

<modify parameter> ::=
<value>
| *<location>*

7.4.6 Connecting an access location

<connect io CHILL simple built-in routine call> ::=
CONNECT (<access location>,<association location>,
<usage expression>[,<where expression>[,<index expression>]])

<usage expression> ::=
<expression>

<where expression> ::=
<expression>

<index expression> ::=
<expression>

7.4.7 Disconnecting an access location

<disconnect io CHILL simple built-in routine call> ::=

DISCONNECT (<access location>)

7.4.8 Accessing attributes of access locations

<access attr io CHILL value built-in routine call> ::=
GETASSOCIATION (<access location>)
| *GETUSAGE (<access location>)*
| *OUTOFFILE (<access location>)*

7.4.9 Data transfer operations

<readrecord io CHILL value built-in routine call> ::=
READRECORD (<access location>[,<index expression>] [,<store location>])

<writerecord io CHILL simple built-in routine call> ::=
WRITERECORD (<access location>[,<index expression>], <write expression>)

<store location> ::=
<static mode location>

<write expression> ::=
<expression>

8 PROGRAM STRUCTURE

8.2 REACHES AND NESTING

<begin-end body> ::=
<data statement list> <action statement list>

<proc body> ::=
<data statement list> { <action statement> | <entry statement>} *

<process body> ::=
<data statement list> <action statement list>

<module body> ::=
{ <data statement> | <visibility statement> | <region> | <spec region> } *
<action statement list>

<region body> ::=
{ <data statement> | <visibility statement>} *

<spec module body> ::=
{ <quasi data statement> | <visibility statement> | <quasi module> | <spec module> |
<quasi region> | <spec region> | <quasi cause statement> } *

<spec region body> ::=
*{ <quasi data statement> | <visibility statement> | <quasi cause statement> }**

<context body> ::=
{ <quasi data statement> | <visibility statement> | <quasi module> | <spec module> |
<quasi region> | <spec region> } *

<quasi module body> ::=
{ <quasi data statement> | <visibility statement> | <quasi module> | <spec module> |
*<quasi region> | <spec region> }**

<quasi region body> ::=
{ <quasi data statement> | <visibility statement> } *

<*action statement list*> ::=
{ <*action statement*>} *

<*data statement list*> ::=
{ <*data statement*>} *

<*data statement*> ::=
<*declaration statement*>
| <*definition statement*>

<*definition statement*> ::=
<*synmode definition statement*>
| <*newmode definition statement*>
| <*synonym definition statement*>
| <*procedure definition statement*>
| <*process definition statement*>
| <*signal definition statement*>
| <*empty*>;

8.3 BEGIN-END BLOCKS

<*begin-end block*> ::=
BEGIN <*begin-end body*> **END**

8.4 PROCEDURE DEFINITIONS

<*procedure definition statement*> ::=
<*defining occurrence*> : <*procedure definition*>
[<*handler*>] [<*simple name string*>];

<*procedure definition*> ::=
PROC ([<*formal parameter list*>]) [<*result spec*>]
[**EXCEPTIONS** (<*exception list*>)] <*procedure attributes*>;
<*proc body*> **END**

<*formal parameter list*> ::=
<*formal parameter*> {,<*formal parameter*>} *

<*formal parameter*> ::=
<*defining occurrence list*> <*parameter spec*>

<*procedure attributes*> ::=
[<*generality*>] [**RECURSIVE**]

<*generality*> ::=
GENERAL
| **SIMPLE**
| **INLINE**

<*entry statement*> ::=
<*defining occurrence*> : <*entry definition*>;

<*entry definition*> ::=
ENTRY

8.5 PROCESS DEFINITIONS

<*process definition statement*> ::=
<*defining occurrence*> : <*process definition*>
[<*handler*>] [<*simple name string*>];

<*process definition*> ::=
PROCESS ([<*formal parameter list*>]); <*process body*> **END**

8.6 MODULES

<*module*> ::=
[<*contexts*>] [<*defining occurrence*>:]
MODULE <*module body*> **END** [<*handler*>] [<*simple name string*>];
| [<*contexts*>] <*remote module*>

8.7 REGIONS

<*region*> ::=
[<*contexts*>] [<*defining occurrence*> :] **REGION** <*region body*> **END**
[<*handler*>] [<*simple name string*>];
| [<*contexts*>] <*remote region*>;

8.8 PROGRAM

<*program*> ::=
{ <*module*> | <*spec module*> | <*region*> | <*spec region*>} +

8.10 CONSTRUCTS FOR PIECEWISE PROGRAMMING

8.10.1 Remote pieces

<*remote module*> ::=
[<*simple name string*>:] **MODULE REMOTE** <*source text designator*>;

<*remote region*> ::=
[<*simple name string*>:] **REGION REMOTE** <*source text designator*>;

<*remote spec module*> ::=
[<*simple name string*>:] **SPEC MODULE REMOTE** <*source text designator*>;

<*remote spec region*> ::=
[<*simple name string*>:] **SPEC REGION REMOTE** <*source text designator*>;

<*remote context*> ::=
CONTEXT REMOTE <*source text designator*> **FOR**

<*source text designator*> ::=
<*character string literal*>
| <*text reference name*>
| <*empty*>

8.10.2 Spec modules, spec regions and contexts

<*spec module*> ::=
[<*contexts*>] [<*simple name string*> :] **SPEC MODULE**
<*spec module body*> **END** [<*simple name string*>];
| <*remote spec module*>

<*spec region*> ::=
[<*contexts*>] [<*simple name string*> :] **SPEC REGION**
<*spec region body*> **END** [<*simple name string*>];
| <*remote spec region*>

<*contexts*> ::=

<*context*> { <*context*> } *

<*context*> ::=
CONTEXT <*context body*> **END** [<*quasi handler*>] **FOR**
| <*remote context*>

8.10.3 Quasi statements

<*quasi data statement*> ::=
<*quasi declaration statement*>
| <*quasi definition statement*>

<*quasi declaration statement*> ::=
DCL <*quasi declaration*> {, <*quasi declaration*> } *;

<*quasi declaration*> ::=
<*defining occurrence list*> <*mode*>
[**STATIC**] [**NONREF**] [**DYNAMIC**]

<*quasi definition statement*> ::=
<*synmode definition statement*>
| <*newmode definition statement*>
| <*synonym definition statement*>
| <*quasi procedure definition statement*>
| <*quasi process definition statement*>
| <*signal definition statement*>
| <*empty*>;

<*quasi procedure definition statement*> ::=
<*defining occurrence*> : **PROC** ([<*quasi formal parameter list*>])
[<*result spec*>] [**EXCEPTIONS** (<*exception list*>)]
<*procedure attributes*> { <*quasi entry statement*> } *
END [<*simple name string*>];

<*quasi entry statement*> ::=
<*defining occurrence*> : **ENTRY** ;

<*quasi formal parameter list*> ::=
<*quasi formal parameter*> {, <*quasi formal parameter*> } *

<*quasi formal parameter*> ::=
[<*simple name string*> {, <*simple name string*> } *] <*parameter spec*>

<*quasi process definition statement*> ::=
<*defining occurrence*> : **PROCESS** ([<*quasi formal parameter list*>])
END [<*simple name string*>];

<*quasi region*> ::=
[<*defining occurrence*> :] **REGION** <*quasi region body*>
END [<*simple name string*>];

<*quasi module*> ::=
[<*defining occurrence*> :] **MODULE** <*quasi module body*>
END [<*simple name string*>];

<*quasi cause statement*> ::=
CAUSE <*exception list*>;

<*quasi handler*> ::=
ON ELSE END
| **ON** <*exception list*> [**ELSE**] **END**

9 CONCURRENT EXECUTION

9.5 SIGNAL DEFINITION STATEMENTS

<signal definition statement> ::=
SIGNAL *<signal definition>* { ,*<signal definition>*} *;

<signal definition> ::=
<defining occurrence> [= (*<mode>* { ,*<mode>*} *)] [**TO** *<process name>*]

10 GENERAL SEMANTIC PROPERTIES

10.1 MODE CHECKING

10.1.3 Case selection

<case label specification> ::=
<case label list> { ,*<case label list>*} *

<case label list> ::=
(*<case label>* { ,*<case label>*} *)
| *<irrelevant>*

<case label> ::=
<discrete literal expression>
| *<literal range>*
| *<discrete mode name>*
| **ELSE**

<irrelevant> ::=
(*)

10.2 VISIBILITY AND NAME BINDING

10.2.4 Visibility in reaches

10.2.4.2 Visibility statements

<visibility statement> ::=
<grant statement>
| *<seize statement>*

10.2.4.3 Prefix rename clause

<prefix rename clause> ::=
(*<old prefix>* -> *<new prefix>*) ! *<postfix>*

<old prefix> ::=
<prefix>
| *<empty>*

<new prefix> ::=
<prefix>
| *<empty>*

<postfix> ::=

<*seize postfix*> {,<*seize postfix*>}*
| <*grant postfix*> {,<*grant postfix*>}*

10.2.4.4 Grant statement

<*grant statement*> ::=
GRANT <*prefix rename clause*> {,<*prefix rename clause*>}*
[[**DIRECTLY**] **PERVASIVE**] ;
| **GRANT** <*grant window*> [<*prefix clause*>]
[[**DIRECTLY**] **PERVASIVE**] ;

<*grant window*> ::=
<*grant postfix*> { , <*grant postfix*> }*

<*grant postfix*> ::=
<*name string*>
| <*newmode name string*> <*forbid clause*>
| [<*prefix*> *!*] **ALL**

<*prefix clause*> ::=
PREFIXED [<*prefix*>]

<*forbid clause*> ::=
FORBID { <*forbid name list*> | **ALL** }

<*forbid name list*> ::=
(<*field name*> { , <*field name*>} *)*

10.2.4.5 Seize statement

<*seize statement*> ::=
SEIZE <*prefix rename clause*> { , <*prefix rename clause*>}*;
| **SEIZE** <*seize window*> [<*prefix clause*>] ;

<*seize window*> ::=
<*seize postfix*> { , <*seize postfix*> }*

<*seize postfix*> ::=
<*name string*>
| <*modulion name string*> **ALL**
| [<*prefix*> *!*] **ALL**

<*modulion name string*> ::=
<*modulion name string*>

11 EXCEPTION HANDLING

11.2 HANDLERS

<*handler*> ::=
ON { <*on-alternative*>}* [**ELSE** <*action statement list*>] **END**

<*on-alternative*> ::=
(<*exception list*>*)* : <*action statement list*>

12 IMPLEMENTATION OPTIONS

12.1 IMPLEMENTATION DEFINED BUILT-IN ROUTINES

<built-in routine call> ::=
    <built-in routine name> ([<built-in routine parameter list>])

<built-in routine parameter list> ::=
    <built-in routine parameter> { , <built-in routine parameter>} *

<built-in routine parameter> ::=
    <value>
  | <location>
  | <non-reserved name>

APPENDIX F: INDEX OF PRODUCTION RULES

non-terminal	defined section	page	used on page(s)
<case label specification>	10.1.3	132	29,33,76
<case selector list>	6.4	76	76
<cause action>	6.12	86	74
<character>	5.2.4.6	53	9,53
<character mode>	3.4.4	18	17
<character string>	2.4	9	9
<character string literal>	5.2.4.6	53	51,54,113
<CHILL built-in routine call>	6.7	83	83
<CHILL directive>	2.6	9	9
<CHILL location built-in routine call>	4.2.12	48	48,83
<CHILL simple built-in routine call>	6.7	83	83
<CHILL value built-in routine call>	5.2.13	62	62,83
<closed dyadic operator>	6.2	75	75
<comment>	2.4	9	
<composite location>	6.5.2	79	79
<composite mode>	3.11.1	26	16
<context>	8.10.2	114	114
<context body>	8.2	105	114
<contexts>	8.10.2	114	111,114
<continue action>	6.15	87	74
<control part>	6.5.1	77	77
<data statement>	8.2	106	105,106
<data statement list>	8.2	105	105
<decimal integer literal>	5.2.4.2	51	51
<declaration>	4.1.1	39	39
<declaration statement>	4.1.1	39	106
<defining mode>	3.2.1	14	14
<defining occurrence>	2.7	10	10,18,19,74,78,91,107,108,110,111 115,121
<defining occurrence list>	2.7	10	14,39,40,41,49,90,107,115
<definition statement>	8.2	106	106
<delay action>	6.16	87	74
<delay alternative>	6.17	88	88
<delay case action>	6.17	88	74
<dereferenced bound reference>	4.2.3	43	41
<dereferenced free reference>	4.2.4	43	41
<dereferenced row>	4.2.5	43	41
<digit>	5.2.4.2	52	8,51,52,53
<directive>	2.6	9	9
<directive clause>	2.6	9	
<discrete mode>	3.4.1	17	16,76
<do action>	6.5.1	77	74
<element layout>	3.11.6	34	27,33
<element mode>	3.11.3	27	27
<else clause>	6.3	76	76
<emptiness literal>	5.2.4.5	53	51
<empty>	6.11	86	86,106,113,115,140,141
<empty action>	6.11	86	74
<end bit>	3.11.6	34	34
<end value>	6.5.2	78	78
<entry definition>	8.4	108	108

non-terminal	defined section	page	used on page(s)
<entry statement>	8.4	108	105
<event length>	3.9.2	24	24
<event list>	6.17	88	88
<event mode>	3.9.2	24	23
<exception list>	3.7	22	22,107,115,147
<exception name>	2.7	10	22,86
<exit action>	6.6	82	74
<expression>	5.3.2	67	18,19,20,24,26,27,29,34,44,45 46,54,61,62,63,66,67,71,76,78 81,83,86,87,98,101,132
<expression conversion>	5.2.11	61	49
<expression list>	4.2.8	45	45,59,62
<field layout>	3.11.6	34	29,32,33
<field name>	2.7	10	55,61,142
<field name defining occurrence>	2.7	10	10
<field name defining occurrence list>	2.7	10	29,32,33
<field name list>	5.2.5	55	55
<fields>	3.11.4	29	29
<first element>	4.2.9	46	46,60
<fixed fields>	3.11.4	29	29
<forbid clause>	10.2.4.4	142	142
<forbid name list>	10.2.4.4	142	142
<for control>	6.5.2	78	77
<formal parameter>	8.4	107	107
<formal parameter list>	8.4	107	107,110
<free directive>	2.6	9	9
<free reference mode>	3.6.3	21	21
<generality>	8.4	107	107
<getstack argument>	5.2.13	62	62
<goto action>	6.9	85	74
<grant postfix>	10.2.4.4	142	141,142
<grant statement>	10.2.4.4	142	140
<grant window>	10.2.4.4	142	142
<handler>	11.2	147	39,40,74,107,110,111
<hexadecimal bit string literal>	5.2.4.7	54	54
<hexadecimal digit>	5.2.4.2	52	52,53,54
<hexadecimal integer literal>	5.2.4.2	52	51
<if action>	6.3	76	74
<implementation directive>			9
<index expression>	7.4.6	98	98,101
<index mode>	3.10.3	25	25,27,33
<initialisation>	4.1.2	39	39
<input-output mode>	3.10.1	25	16
<instance mode>	3.8	23	16
<integer literal>	5.2.4.2	51	51
<integer mode>	3.4.2	17	17
<io CHILL location built-in routine call>	7.4.2	96	48,95
<io CHILL simple built-in routine call>	7.4.9	101	83,95
<io CHILL value built-in routine call>	7.4.9	101	62,95,96

non-terminal	defined section	page	used on page(s)
<irrelevant>	10.1.3	132	132
<iteration>	6.5.2	78	78
<labelled array tuple>	5.2.5	54	54
<labelled structure tuple>	5.2.5	55	55
<left element>	4.2.7	44	44,58
<length>	3.11.6	34	34
<letter>	5.2.4.6	53	8,53
<level structure mode>	3.11.5	32	28
<lifetime-bound initialisation>	4.1.2	39	39
<literal>	5.2.4.1	51	49
<literal expression list>	3.11.4	29	29
<literal range>	3.4.6	19	19,25,132
<location>	4.2.1	41	40,44,45,46,47,48,50,62,63,72 74,79,81,83,85,86,87,88,89,90 91,96,97,98,100,101,149
<location built-in routine call>	4.2.12	48	41
<location contents>	5.2.2	50	49
<location conversion>	4.2.13	48	41
<location declaration>	4.1.2	39	39
<location enumeration>	6.5.2	78	78
<location procedure call>	4.2.11	47	41
<loc-identity declaration>	4.1.3	40	39
<loop counter>	6.5.2	78	78,79
<lower bound>	3.4.6	19	19
<lower element>	4.2.9	46	46,60
<map reference name>	2.7	11	34
<member mode>	3.5	20	20
<membership operator>	5.3.4	68	68
<mode>	3.3	16	14,20,21,22,24,25,27,29,32,33 39,40,41,49,76,78,115,121
<mode definition>	3.2.1	14	15,16
<modify parameter>	7.4.5	97	97
<modify parameter list>	7.4.5	97	97
<module>	8.6	111	74,112
<module body>	8.2	105	111
<modulion name string>	10.2.4.5	144	144
<monadic operator>	5.3.7	71	71
<multiple assignment action>	6.2	74	74
<name>	2.7	10	17,18,19,20,21,22,23,24,25,26 27,29,41,42,43,47,48,50,52,54 61,62,63,65,66,83,89,90,121,132 149
<name string>	2.7	10	10,142,144
<nested structure mode>	3.11.4	29	28
<newmode definition statement>	3.2.3	16	106,115
<new prefix>	10.2.4.3	140	140
<(n) level alternative>	3.11.5	33	33
<(n) level alternative fields>	3.11.5	32	32
<(n) level fields>	3.11.5	32	
<(n) level fixed fields>	3.11.5	32	32

non-terminal	defined section	page	used on page(s)
<(n) level variant fields>	3.11.5	33	33
<non-composite mode>	3.3	16	16
<numbered set element>	3.4.5	18	18
<numbered set list>	3.4.5	18	18
<(n+1) level fields>			32,33
<octal bit string literal>	5.2.4.7	54	54
<octal digit>	5.2.4.2	52	52,53,54
<octal integer literal>	5.2.4.2	51	51
<old prefix>	10.2.4.3	140	140
<on-alternative>	11.2	147	147
<operand-1>	5.3.3	68	67,68
<operand-2>	5.3.4	68	68
<operand-3>	5.3.5	69	68,69
<operand-4>	5.3.6	71	69,71
<operand-5>	5.3.7	71	71
<operand-6>	5.3.8	72	71
<operator-3>	5.3.4	68	68
<operator-4>	5.3.5	69	69
<origin array mode name>	3.11.3	27	27
<origin string mode name>	3.11.2	26	26
<origin variant structure mode name>	3.11.4	29	29
<parameter attribute>	3.7	22	22
<parameterised array mode>	3.11.3	27	27
<parameterised string mode>	3.11.2	26	26
<parameterised structure mode>	3.11.4	29	28
<parameter list>	3.7	22	22
<parameter spec>	3.7	22	22,107,115
<parenthesised expression>	5.2.16	66	50
<pos>	3.11.6	34	34
<postfix>	10.2.4.3	141	140
<powerset difference operator>	5.3.5	70	69,75
<powerset enumeration>	6.5.2	78	78
<powerset inclusion operator>	5.3.4	69	68
<powerset mode>	3.5	20	16
<powerset tuple>	5.2.5	54	54
<prefix>	2.7	10	10,140,142,144
<prefix clause>	10.2.4.4	142	142,144
<prefixed name string>	2.7	10	10,11
<prefix rename clause>	10.2.4.3	140	142,144
<primitive value>	5.2.1	49	43,58,59,60,61,63,72,81,83,89
<priority>	6.16	87	87,88,89
<proc body>	8.2	105	107
<procedure attributes>	8.4	107	107,115
<procedure call>	6.7	83	47,62,83
<procedure definition>	8.4	107	107
<procedure definition statement>	8.4	107	106
<procedure mode>	3.7	22	16
<process body>	8.2	105	110
<process definition>	8.5	110	110
<process definition statement>	8.5	110	106
<program>	8.8	112	

non-terminal	defined section	page	used on page(s)
<quasi cause statement>	8.10.3	115	105
<quasi data statement>	8.10.3	114	105
<quasi declaration>	8.10.3	115	115
<quasi declaration statement>	8.10.3	114	114
<quasi definition statement>	8.10.3	115	114
<quasi entry statement>	8.10.3	115	115
<quasi formal parameter>	8.10.3	115	115
<quasi formal parameter list>	8.10.3	115	115
<quasi handler>	8.10.3	115	114
<quasi module>	8.10.3	115	105
<quasi module body>	8.2	105	115
<quasi procedure definition statement>	8.10.3	115	115
<quasi process definition statement>	8.10.3	115	115
<quasi region>	8.10.3	115	105
<quasi region body>	8.2	105	115
<range>	5.2.5	54	54
<range enumeration>	6.5.2	78	78
<range list>	6.4	76	76
<range mode>	3.4.6	19	17
<reach-bound initialisation>	4.1.2	39	39
<receive buffer case action>	6.19.3	91	90
<receive case action>	6.19.1	90	74
<receive expression>	5.3.8	72	72
<receive signal case action>	6.19.2	90	90
<record mode>	3.10.3	25	25
<referenced location>	5.3.8	72	72
<referenced mode>	3.6.2	21	21
<reference mode>	3.6.1	21	16
<region>	8.7	111	105,112
<region body>	8.2	105	111
<register name>	2.7	10	22
<relational operator>	5.3.4	68	68
<remote context>	8.10.1	113	114
<remote module>	8.10.1	113	111
<remote region>	8.10.1	113	111
<remote spec module>	8.10.1	113	114
<remote spec region>	8.10.1	113	114
<result>	6.8	85	85
<result action>	6.8	85	74
<result attribute>	3.7	22	22
<result spec>	3.7	22	22,107,115
<return action>	6.8	85	74
<right element>	4.2.7	44	44,58
<row mode>	3.6.4	22	21
<seize postfix>	10.2.4.5	144	141,144
<seize statement>	10.2.4.5	144	140
<seize window>	10.2.4.5	144	144
<send action>	6.18.1	88	74
<send buffer action>	6.18.3	89	88
<send signal action>	6.18.2	89	88

non-terminal	defined section	page	used on page(s)
<set element>	3.4.5	19	19
<set list>	3.4.5	18	18
<set literal>	5.2.4.4	52	51
<set mode>	3.4.5	18	17
<signal definition>	9.5	121	121
<signal definition statement>	9.5	121	106,115
<signal receive alternative>	6.19.2	90	90
<simple name string>	2.2	8	9,10,11,74,82,85,107,110,111,113 114,115
<simple name string list>	2.6	9	9
<simple prefix>	2.7	10	10
<single assignment action>	6.2	74	74
<slice size>	4.2.7	44	44,46,59,60
<source text designator>	8.10.1	113	113
<space>	5.2.4.6	53	53
<spec module>	8.10.2	114	74,105,112
<spec module body>	8.2	105	114
<spec region>	8.10.2	114	105,112
<spec region body>	8.2	105	114
<start action>	6.13	86	74
<start bit>	3.11.6	34	34
<start element>	4.2.7	44	44,58,59
<start expression>	5.2.14	65	50,86
<start value>	6.5.2	78	78
<step>	3.11.6	34	34
<step enumeration>	6.5.2	78	78
<step size>	3.11.6	34	34
<step value>	6.5.2	78	78
<stop action>	6.14	87	74
<store location>	7.4.9	101	101
<string concatenation operator>	5.3.5	70	69
<string element>	4.2.6	44	41
<string length>	3.11.2	26	26
<string mode>	3.11.2	26	26
<string repetition operator>	5.3.7	71	71
<string slice>	4.2.7	44	41
<string type>	3.11.2	26	26
<structure field>	4.2.10	47	41
<structure mode>	3.11.4	28	26
<structure tuple>	5.2.5	54	54
<sub expression>	5.3.2	67	67
<sub operand-1>	5.3.3	68	68
<sub operand-2>	5.3.4	68	68
<sub operand-3>	5.3.5	69	69
<sub operand-4>	5.3.6	71	71
<symbol>	5.2.4.6	53	53
<synchronisation mode>	3.9.1	23	16
<synmode definition statement>	3.2.2	15	106,115
<synonym definition>	5.1	49	49
<synonym definition statement>	5.1	49	106,115
<tags>	3.11.4	29	29,33
<text reference name>	2.7	11	113

non-terminal	defined section	page	used on page(s)
<then clause>	6.3	76	76
<tuple>	5.2.5	54	49
<undefined value>	5.3.1	66	66
<unlabelled array tuple>	5.2.5	54	54
<unlabelled structure tuple>	5.2.5	55	55
<unnamed value>	3.4.5	19	19
<unnumbered set list>	3.4.5	18	18
<upper bound>	3.4.6	19	19
<upper element>	4.2.9	46	46,60
<upper index>	3.11.3	27	27
<upper lower argument>	5.2.13	62	62
<usage expression>	7.4.6	98	98
<value>	5.3.1	66	39,49,54,55,62,74,83,85,89,96 97,149
<value array element>	5.2.8	59	49
<value array slice>	5.2.9	60	49
<value built-in routine call>	5.2.13	62	49
<value enumeration>	6.5.2	78	78
<value name>	5.2.3	50	49
<value procedure call>	5.2.12	62	49
<value string element>	5.2.6	58	49
<value string slice>	5.2.7	58	49
<value structure field>	5.2.10	61	49
<variant alternative>	3.11.4	29	29
<variant fields>	3.11.4	29	29
<visibility statement>	10.2.4.2	140	105
<where expression>	7.4.6	98	98
<while control>	6.5.3	81	77
<with control>	6.5.4	81	81
<with part>	6.5.4	81	78
<word>	3.11.6	34	34
<write expression>	7.4.9	101	101
<zero-adic operator>	5.2.15	66	50

APPENDIX G: INDEX

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

PART II

Supplements to Recommendation Z.200

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

LIST OF CHILL TRAINING DOCUMENTS

CCITT Study Group XI, period 1977-1980: Introduction to CHILL (in English). A CCITT Manual, prepared by Study Group XI.

Available through: ITU Sales Section, Place des Nations, CH-1211 Geneva 20, Switzerland.

Introduction to CHILL (in Japanese). K. Maruyama, NTT, Tokyo, May 1981.

Available through: Mr. Norio Sato, Musashino-ECL, Nippon Telegraph and Telephone, Musashino-shi, Tokyo 180, Japan.

Introduction to the CCITT high level language for SPC systems: CHILL (in English). W. Buerger, H. Sorgenfrei, Siemens Telefone System Division, 1978. A collection of copies from transparencies.

Not available through normal book service.

Management information on CHILL. H. Sorgenfrei, Siemens Telefone Systems Division, 1980. A collection of copies from transparencies.

Not available through normal book service.

A software development system based on CHILL. H. Krafka, Siemens OeV ET S3. A 3-page report plus copies from transparencies.

Not available through normal book service.

Introduzione all linguaggio CHILL (in Italian). R. Martucci, ITALTEL. Progetto Finalizzato Informatica CNR, ETS/PISA 1982.

Available through: CSELT. Via Reiss Romoli 274, I-10148 Torino, Italy.

CHILL: Die Neue CCITT-sprache (in German). H. Zwittlinger, Abend-Technikum und Software-Schule, Bern, Switzerland.

H. Lang & Cie., Bern 1981 (Vol. 4, Beiträge Mathematik, Informatik, Nachrichtentechnik). A course consisting of 16 tables, held at University Bern 1980/81.

Available through bookshops.

CHILL: the standard language for SPC systems. K. Rekdal, RUNIT, Trondheim, Norway. A collection of copies from computer-print transparencies.

Available through: Mr. K. Rekdal, RUNIT, Strindvegen 2, N-7034 Trondheim, Norway.

CHILL/D: A self-instructional manual (Volumes I, II, III). T. Valk-Fai (Philips PITTC), Philips PTI, Hilversum, 1982. A course document covering more than the indicated subset.

Order from: M. J. van Doggenaar, PITTC, PO Box 32, 1200 JD Hilversum, Netherlands.

CHILL: Eine moderne Programmiersprache für die Systemtechnik. W. Sammer, H. Schwaertzel (Siemens AG), Springer Verlag, 1982. Handbook on CHILL and description of the Siemens Implementation (in German).

Available through bookshops.

Elementos del Lenguaje CHILL. CTNE (J. Munera, editor), Madrid, Mayo 1982. Course material to be extended describing a basic subset of CHILL (in Spanish).

Available through: Mr. J. Munera, CTNE, Dept. de Normativa Técnica, Apartado 753, Madrid 13, Spain.

Einführung in die CCITT High Level Programming Language CHILL. W. Buerger, H. Sorgenfrei, W. Eldon, Siemens AG, Bereich Fernsprechsysteme, München, Mai 1980. A course with tasks and examples (in German), 3 volumes.

Available through: Siemens Lehr- und Lernmittel, Postfach 830451, D-8000 Munich 83, F.R. Germany.

CCITT Study Group XI, period 1980-1981. Rapporteur for CHILL Maintenance: CHILL User Manual (5th draft). A teachers and Students handbook on CHILL.

Not yet available through ITU Sales Service, but published in the CHILL Bulletin, Vol. 4, No. 1, March 1984.

Available through: Mr. Kees Smedema, AT&T and Philips Telecommunications, Rue des Deux-Gares 80, B-1070, Brussels, Belgium.

Supplement No. 2

LIST OF ACCESSES TO CHILL PROGRAMMING SYSTEMS FOR NON-PROFIT USE BY SCIENTIFIC AND EDUCATIONAL BODIES

(Up to October 1984)

Siemens: The compiler plus debugger may be provided to scientific or educational users, a fee will be raised to cover transfer expenses. An implementation at the Technical University of Berlin has been accomplished.

Contact: Mr. Reithmaier, Siemens AG, K OeV EP D13, POB 700073, D-8000 Munich 70, F.R. Germany.

STERIA: Price of a source level licence for educational or scientific use of the CHILL front-end is 300 000 FF.

Contact: STERIA, avenue de l'Europe, F-78140 Vélizy-Villacoublay, France.

PTT Dr. Neher Laboratory: The CHILL front-end plus executing virtual machine and debugger may be used by scientific and educational institutions in the Netherlands. Computer time will be charged.

Contact: Mr. G. H. te Sligte, Dr. Neher Laboratory, POB 421, NL-2260 Leidschendam, Netherlands.

Nordic CHILL compiler: The CHILL integrated programming system CHIPSY may be accessed by scientific and educational institutions against a nominal charge. The programming system, on the Hasler Bern installation, is being used by students of the Software-School Switzerland and the informatics department of the University of Bern.

Contact: RUNIT, Strindvegen 2, N-7034 Trondheim, Norway.

Danish Telecom Research Laboratory: The CHILL compiler and interpreter can be obtained by scientific and educational bodies at a nominal charge.

Contact: Peter Haff, Telecom Research Laboratory, Borups Alle 43, DK-2200 Copenhagen N, Denmark.

NTT, Japan: Method and conditions for access to the CHILL programming system are being investigated.

Supplement No. 3

LIST OF REFERENCES TO PUBLICATIONS ABOUT CHILL

ANDERSEN (T.): A Portable CHILL Runtime System. Scandpower. *2nd CHILL Conference*, Lisle, Illinois, USA. 10 pages, 1983.

BORDELON (E. P.): Name Binding in CHILL. Bell Laboratories. *2nd CHILL Conference*, Lisle, Illinois, USA. 10 pages, 1983.

BOTSCH (D.): The use of high level language programming and its impact on the software of digital switching systems. *ISS'79*, Paris, 1979.

BOURGONJON (R. H.): CHILL... The standard high level language for programming SPC telephone exchanges. Philips' Telecommunicatie Industrie B.v., Hilversum, Netherlands. *Journal A*, Volume 22, No. 4, 4 pages in English, 1981.

BOURGONJON (R. H.), BREEUS (C.): Implementation experience with the CCITT high level language CHILL. *ISS'79*, Paris, 1979.

BOURGONJON (R. H.): The CCITT high level programming language. *3rd SETSS Conference*, Helsinki, Finland, 1978.

BOURGONJON (R. H.), REKDAL (K.): CHILL user's manual, 1981.

BOURGONJON (R. H.): Programming Languages, Environments and CHILL. Philips' Telecommunicatie Industrie B.v., Netherlands. *2nd CHILL Conference*, Lisle, Illinois, USA. 6 pages, 1983.

BOUTE (R. T.), JACKSON (M. I.): A joint evaluation of the programming language ADA and CHILL. BTM, Antwerpen, Belgium. Standard Telecom. Laboratory, Harlow, United Kingdom. *IEE Conference*, Publication No. 198, 6 pages in English, 1981.

BRANQUART (P.), LOUIS (G.), WODON (P.): Aspects de CHILL, le langage du CCITT. MBLE, Bruxelles, Belgium. *Revue TSI (Technique et Science informatique)*, 9 pages in French, 1982.

BRANQUART (P.), LOUIS (G.), WODON (P.): On the Analytical Description of CHILL. Philips Research Laboratory, Brussels. *2nd CHILL Conference*, Lisle, Illinois, USA. 6 pages, 1983.

BUTCHER (B. A.): Selecting an Appropriate CHILL Subset. ITT-ATC. *2nd CHILL Conference*, Lisle, Illinois, USA. 6 pages in English, 1983.

CAIN (G. J.), JACKSON (L. N.), VESETAS (R.), WALTER (A.), YONG (W. B.): Computer aided software generation (the MELBA system for generating CHILL code). *4th SETSS Conference*, University of Warwick, Coventry, United Kingdom, 1981.

CAMICI (A.), GIARRATANA (V.), NIRO (F.), MODESTI (M.): CHILL for supporting software engineering environments. CSELT & SIP, Italy. *5th SETSS Conference*, Lund, Sweden. 4 pages in English, 1983.

CAMICI (A.), GIARRATANA (V.), NIRO (F.), PANARONI (P.): Criteri di progetto nell'implementazione del linguaggio di programmazione CHILL. *Congresso AICA 1980*, Bologna, 1980.

CAMICI (A.), GIARRATANA (V.), MANUCCI (F.): A CHILL software development system for distributed architecture. *ICC'81*, Denver, USA, 1981.

CARRELLI (C.), MANUCCI (F.), ROSCI (G.): CHILL Programming System: implementation and operational aspects. *Congress ISS'81'CIC Symposium*, Montreal, Canada (in English), 1981.

CARRELLI (C.), MANUCCI (F.), MARTUCCI (R.): The CCITT high level language: an approach in Italy. *ISS'79*, Paris, 1979.

CCITT: CHILL language definition. *Recommendation Z.200*, 1984.

CONROY (R. A.): Impacts of CHILL on System Design. ITT-ATC. *2nd CHILL Conference*, Lisle, Illinois, USA. 3 pages, 1983.

DACKER (B.), JACOBSON (I.): Real time system design using CHILL. *3rd SETSS Conference*, Helsinki, Finland, 1978.

DE BACHTIN (O.), LINDROOS (L.), TONNBY (I.): Programmed testing of AXE systems using a CHILL based language PILOT. *4th SETSS Conference*, University of Warwick, Coventry, United Kingdom, 1981.

DENENBERG (C. G.): CHILL implementation techniques. *3rd SETSS Conference*, Helsinki, Finland, 1978.

DE NICOLA (R.), MARTUCCI (R.), ROBERTI (P.): A CHILL based distributed architecture. *International Computing Symposium*, London, 1981.

DENIS (G.), LANGLOIS (C.): Presentation d'une machine langage orientée vers le langage CHILL. CNET, Paris, France. *Revue L'écho des Recherches*, 7 pages in French, 1981.

DENIS (G.), LANGLOIS (C.), D'ISSERNIO (J. P.): Machine langage adaptée à CHILL: premiers résultats d'évaluation. CNET, Paris, France. *Congrès ISS'81'CIC Symposium*, Montreal, Canada. 6 pages in English and in French, 1981.

FEICHT (E. J.): "CHILL Factory": production and maintenance of a large CHILL software system. Siemens, F.R. Germany. *5th SETSS Conference*, Lund, Sweden. 6 pages in English, 1983.

GIARRATANA (V.), MODESTI (M.): An SDL into CHILL skeleton translation system. CSELT & SIP, Italy. *5th SETSS Conference*, Lund, Sweden. 6 pages in English, 1983.

GIARRATANA (V.), GIANNETI (B.), MUSSA (P. L.): Verso la formalizzazione della generazione di codice nei compilatori: un generatore di codice indipendente della macchina per il CHILL. *Congresso AICA 1980*, Bologna, 1980.

GREEN (G. A.), HALLSTEINSEIN (S. O.), WANVIK (D. H.), NOKKEN (L.): Separate Compilation in CHIPSY. RUNIT. STK. *2nd CHILL Conference*, Lisle, Illinois, USA, 1983.

GUTFELDT (H.): Modelling telecom processes with CHILL process. Hassler AG, Bern, Switzerland. *5th SETSS Conference*, Lund, Sweden. 4 pages in English, 1983.

GUTFELDT (H.): CHILL. Hassler AG, Bern, Switzerland. *Hassler Werk Zeitung*, 1 page in German, 1981.

GUTFELDT (H.): SDL and CHILL Structured Programming. Hassler AG, Bern, Switzerland. *2nd CHILL Conference*, Lisle, Illinois, USA. 13 pages, 1983.

GUTTMAN (N.): Efficient Implementation of Nested Non-recursive Procedures in CHILL. Bell Laboratories. *2nd CHILL Conference*, Lisle, Illinois, USA. 6 pages, 1983.

HAFF (P.), BJOERNER (D.): CHILL Formal definition. Dansk Datamatik Center, Lyngby, Denmark, 1981.

HAMMER (D.), FRANKEN (G.), GREEN (P. C): A distributed Operating System for the TCP16 System. Philips' Telecommunicatie Industrie B.v., Netherlands. *5th SETSS Conference*, Lund, Sweden. 6 pages in English, 1983.

HAQUE (T. A.), DALEY (R. W.): ITT 1240 Digital Exchange – CHILL Programming environment. ITT-ESC, 1983.

HAQUE (T. A.): ITT CHILL Programming environment, *Electrical Communication*, Vol. 9. ITT-ESC, 1983.

HAQUE (T. A.): CHILL Programming environment, *Proceedings of COMPSAC'83*, Chicago, USA. ITT-ESC, 1983.

HRVENSALO (J.): CHILL – unsi standardikieli. Finnish state research center, Finland. *Elektroniikka No. 19*, 3 pages in Finnish, 1981.

KEEDY (J. L.): A report on the concurrent processing features of the CCITT language CHILL. Telecom Australia. 45 pages in English, 1981.

KRAFKA (H. H.): A software development system based on CHILL. *4th SETSS Conference*, University of Warwick, Coventry, United Kingdom, 1981.

KURKI-SUONIO REINO: Mikrojen uudet. University of Tampere, Finland. *Elektroniikka No. 19*, 3 pages in Finnish, 1981.

LANGLOIS (C.): Evaluation of a CHILL Oriented Processor. CNET. *2nd CHILL Conference*, Lisle, Illinois, USA. 9 pages, 1983.

LO (P.), SHAW (F.): A Distributed Operating System for CHILL. ITT-ATC. *2nd CHILL Conference*, Lisle, Illinois, USA. 5 pages, 1983.

McCULLOUGH (R. H.): A CHILL Compiler Based on the Portable C Compiler. Bell Laboratories. *2nd CHILL Conference*, Lisle, Illinois, USA. 13 pages, 1983.

MARUYAMA (K.), KONISHI (K.), SATO (N.): NTT CHILL Implementation Aspects and its Application Experience. Musashino electrical communication laboratory, NTT, Japan. 6 pages in English, 1981.

MEIJER (R. W.), te SLIGTE (G. H.): Status report of CCITT HLL implementation at the Dr. Neher Laboratory of the Netherlands PTT. *3rd SETSS Conference*, Helsinki, Finland, 1978.

MEILING (E.), STEEN (U.), PALM: A comparative Study of CHILL and ADA on the Basis of Denotational Descriptions. Dansk Datamatik Center, Lyngby. *2nd CHILL Conference*, Lisle, Illinois, USA. 14 pages, 1983.

MODESTI (M.), GIARRATANA (V.): An SDL to CHILL Skeleton Transformer. SIP DG Roma, CSELT Torino. *2nd CHILL Conference*, Lisle, Illinois, USA. 13 pages, 1983.

MOORE (B. G.), CHANDRASEKHARAN (M.): Tools for maintaining consistency in large programs compiled in parts. GTE Laboratories, USA. *5th SETSS Conference*, Lund, Sweden. 5 pages in English, 1983.

OLSEN (N. C.): Alternatives for handling I/O in CHILL. ITT-ATC. *2nd CHILL Conference*, Lisle, Illinois, USA. 5 pages, 1983.

PANARONI (P.), RUGANI (U.): Il parallelismo nel linguaggio CHILL: descrizione, analisi comparata e sua implementazione. *Congresso AICA 1980*, Bologna, 1980.

REITHMAIER (E.): Compilation Control in a large CHILL Application. Siemens. *2nd CHILL Conference*, Lisle, Illinois, USA. 10 pages, 1983.

RIETSCHOTE van (H. F.): Debugging in a CHILL Oriented Program Development System. Philips' Telecommunicatie Industrie B.v., Netherlands. *2nd CHILL Conference*, Lisle, Illinois, USA. 4 pages, 1983.

REKDAL (K.): CHILL, the standard language for programming SPC systems. *IEEE Transactions on Telecommunications*, June 1981. Proceedings of NTC, New Orleans, USA, November 1981. 20 pages in English, 1981.

REKDAL (K.): CHILL, the standard language for programming SPC systems. Mini-micro systems, Shenyang, China. 7 pages in Chinese, 1982.

REKDAL (K.): Introduction to CHILL, 1980.

REKDAL (K.), BOTNEVIK (H.), HALLSTEINSEIN (S. O.), VENSTAD (A.): Some implementation aspects of CHILL. *3rd SETSS Conference*, Helsinki, Finland, 1978.

REKDAL (K.): CHILL in the Software Engineering Context. *5th SETSS Conference*, Lund, Sweden, 1983.

REKDAL (K.): CHILL, The Standard Language. *Proceedings of COMPSAC'83*, Chicago, USA, 1983.

RUDMIK (A.), MOORE (B. G.): The separate Compilation of Very Large CHILL Programs. GTE. *2nd CHILL Conference*, Lisle, Illinois, USA. 11 pages, 1983.

SMEDEMA (C. H.), BISHOP (R.), CHEUNG (R. C. H.), BORDELON (E. P.), FEAY (M. R.), LOUIS (G.): Separate Compilation and the development of large programs in CHILL. PTI Netherlands, British Telecom UK, Bell Laboratories USA and PRL Belgium. *5th SETSS Conference*, Lund, Sweden. 7 pages in English, 1983.

SMEDEMA (C. H.), MEDEMA (P.), BOASSON (M.): The programming languages: PASCAL, MODULA, CHILL, ADA. Prentice-Hall International. 160 pages in English, 1983.

SMEDEMA (C. H.): A test approach for CHILL based systems. Philips' Telecommunicatie Industrie B.v., Hilversum, Netherlands. *2nd CHILL Conference*, Lisle, Illinois, USA. 4 pages, 1983.

TEICHROEW (D.), BLOCK (C.), KYO CHUL KANG, CHIKOFSKY (E.): Usage of the System Encyclopedia Manager (SEM) System with the CCITT Functional Specification and Description Language (CCITT/SDL). *2nd CHILL Conference*, Lisle, Illinois, USA. 15 pages, 1983.

THALHAMER (J. A.): Design Issues of a High Level Symbolic Debugger for CHILL. ITTE-PSC, 1983.

THEURETZBACHER (N.): Implementation of the CHILL Tasking Concept in a Compiler and a Real Time Operating System. ITT-Austria. *2nd CHILL Conference*, Lisle, Illinois, USA. 6 pages, 1983.

VALK-FAI (T.): A training course in the use of CHILL. PTI. *2nd CHILL Conference*, Lisle, Illinois, USA. 11 pages, 1983.

VENSTAD (A.): On rehosting CHIPSY. RUNIT. *2nd CHILL Conference*, Lisle, Illinois, USA. 10 pages, 1983.

WEN (W.): Problem Oriented Languages. ITT-ATC. *2nd CHILL Conference*, Lisle, Illinois, USA. 4 pages, 1983.

WINKLER (J. F. H.): A new Methodology for I/O and its application to CHILL. Siemens AG, Munich. *2nd CHILL Conference*, Lisle, Illinois, USA. 16 pages, 1983.

ZWITTLINGER (Dr. H.): CHILL, die neue CCITT Sprache. Ingenieurschule, Bern, Switzerland. Programmiersprachen für die Nachrichtentechnik, University Bern. 76 pages in German, 1980/81.

LIST OF CHILL IMPLEMENTATIONS AND APPLICATIONS

Origin		CHILL compiler				CHILL applications			
Country	Organization	Host computer	OS	Target computere	Status [1]	SPC system	Support software	Others	Status [1]
Federal Republic of Germany	Siemens AG	7700	BS 1000/ BS 2000	7700	I	EWSD,			I
				7800	I	ETS,	"Complete" development and production system		I
		7800	BS3000	IBM 370	I				
				SSP 103	I	BIGFON,			I
		IBM 370	OS/MVS	SSP 112D	I	EMS			U
				8086 Family	I				
						ISDN			U
	Tekade	VAX II	VMS (Front end)	6800 (Front end)	U U				
	Standard Electrik Lorenz (ITT/SEL)	IBM 370	MVS	iAPX 86	I	System 12			I
		IBM 370	MVS	8086 Family	I	5600 BCS	Development and production system		U
									I
						ISDN			U
						DFS			U

[1] P: Planned.
U: Under development.
I: Implemented.

LIST OF CHILL IMPLEMENTATIONS AND APPLICATIONS *(cont.)*

Origin		CHILL compiler				CHILL applications			
Country	Organization	Host computer	OS	Target computer	Status [1]	SPC system	Support software	Others	Status [1]
Austria	ITT Austria	HP VAX	 VMS	8085 8086	I I	5200 BCS (AMANDA)			I
Belgium	BTM	IBM 370	MVS	8086	I	System 12 ISDN	 Development and production system		I I U
Brazil	CPqD TELEBRAS	VAX 11	VMS	iAPX 286	P	TROPICO			P
Denmark	Telecom Research Lab	VAX (portable)	VMS	Virtual machine hosted on VAX (portable)	I (Comercially available)		Symbolic CHILL Debugger		U

[1] P: Planned.
U: Under development.
I: Implemented.

LIST OF CHILL IMPLEMENTATIONS AND APPLICATIONS *(cont.)*

Origin		CHILL compiler				CHILL applications			
Country	Organization	Host computer	OS	Target computer	Status [1]	SPC system	Support software	Others	Status [1]
Spain	CTNE	No CHILL compiler planned until now *Note* – We are using ITX tools in SESA (SESARC) and we have no decision for the future about tools to be used.				System 12			U
United States of America	ATT-Bell Labs.				P				
	ITT	IBM 370	MVS	8086	I	System 12	Development and production system		I I
	CTE Laboratories	IBM 370 VAX 11/780 TANDEM	IBM 370 based OS	IBM 370 TANDEM VAX 11/780 I 8086 I 432	I I I I U				

[1] P: Planned.
U: Under development.
I: Implemented.

LIST OF CHILL IMPLEMENTATIONS AND APPLICATIONS *(cont.)*

Origin		CHILL compiler				CHILL applications			
Country	Organization	Host computer	OS	Target computer	Status [1]	SPC system	Support software	Others	Status [1]
Finland	Technical Research Centre, Telecommunications Laboratories						CHILL run-time support software for compiled programs	UMC micro-computer	U
France	PTT and STERIA	DPS 8	MULTICS	CHILL Processor FRONT END	I I	Experiment of SPC system			I
Italy	TELETTRA	UNIVAC 1100/60	EXEC 8	MIC 30 MIC 10	U				
	SIP-CSELT	VAX 780 VAX 780 VAX 780	VMS UNIX VMS	MIC 20 VAX 780 VAX 780	I U I				
	ITALTEL-SIT SPA	VAX 11/750	MVS	MIC 20	U	UT-100 new switching system	Optimizing processor working on the output of the compiler SDL-like language to CHILL Translator		I I
	FACE (ITT)	IBM 370	MVS	8086 Family	I	System 12	 Development and production system		I I

[1] P: Planned.
U: Under development.
I: Implemented.

LIST OF CHILL IMPLEMENTATIONS AND APPLICATIONS *(Cont.)*

Origin		CHILL compiler				CHILL applications			
Country	Organization	Host computer	OS	Target computer	Status [1]	SPC system	Support software	Others	Status [1]
Japan	KDD	D10 (NTT's compiler)	D180	D10	I	XE-10 Digital INTS XE-20 Digital INTS			U U
	NTT	D10	D180	D10	I	– D10 ESS (D100B, Mobile telephone, data telephone)	Office data generator (see SETSS 1975 paper)		I
						– D60 (Digital telephone transit)			I
		DIPS	DIPS (NTT's TSS and real-time OS)	D10 D10-VLSI	I	– Facsimile storage and conversion system			I
						– D70 (Digital telephone combined transit local)			I
						– D50 (Circuit data switching)			U
						– Wideband exchange system			U
		DIPS	DISP	D10 and D10-VLSI	U [2]	– D70 version-up INS			U U

[1] P: Planned.
U: Under development.
I: Implemented.

[2] Enhanced with partial code generation and module decomposition (see second CHILL conference paper).

LIST OF CHILL IMPLEMENTATIONS AND APPLICATIONS *(cont.)*

Origin		CHILL compiler				CHILL applications			
Country	Organization	Host computer	OS	Target computer	Status [1]	SPC system	Support software	Others	Status [1]
Norway	STK (ITT)	ND100	SINTRAN III	iAPX 86	I	5500 BCS Digimat 2000			I I
	Norwegian Telecom Administration	ND 100	SINTRAN III	I 8086	I	X25 Costal radio-control Experimental National N/& O Network Experimental service integrated video-switch			U U P P
Netherlands	Administration	PDP 10	(TOPS-10 (TOPS-20	PDP-11 MC 68000 PDP-10	I I U		Support system PRX/A OMC	OS Signalling C7 ISDN D-channel protocol Data base application Terminal multiplexes	I I U U U P I I
	AT&T and Philips Telecom	IBM 370	UNIX	IBM 370	U				

[1] P: Planned.
U: Under development.
I: Implemented.

LIST OF CHILL IMPLEMENTATIONS AND APPLICATIONS *(end)*

Origin		CHILL compiler				CHILL applications			
Country	Organization	Host computer	OS	Target computer	Status [1]	SPC system	Support software	Others	Status [1]
United Kingdom	BT	ND 100	SINTRAN	8086	I			Teletext Protocol	I P
	ITT/ESC	IBM 370	MVS	8086	I	System 12 5600 BCS	Support Software		I U
Switzerland	Hasler	ND-100 (compiler delivered by RUNIT)	SINTRAN III	I 8086	I		Planned Target OS	Telex Teletex Convertor MARK II	 U U
USRR	Moscow Institute of Telecommunications	ES-1033 CC-NEVA-1M	OS ES NEVA-1M	CC NEVA-1M	I				
	Moscow State University	BESM-6		CC NEVA-1M	I				
Denmark Finland Norway Sweden United Kingdom	RUNIT Nordic Telecom Administration and British Telecom	ND 100	SINTRAN III	iAPX 86 iAPX 286 ND 100	I U U				
		Intellec MDS	ISIS II CP/M-86	iAPX 86 iAPX 86	I I				
		VAX 11	VMS UNIX	iAPX 86 iAPX 86	P U		CHILL run time system		I
		IBM PC	CP/M-86	iAPX 86	I				

[1] P: Planned.
U: Under development.
I: Implemented.

LIST OF CHILL IMPLEMENTATIONS AND APPLICATIONS *(cont.)*

Origin		CHILL compiler				CHILL applications			
Country	Organization	Host computer	OS	Target computer	Status [1]	SPC system	Support software	Others	Status [1]
Norway	STK (ITT)	ND100	SINTRAN III	iAPX 86	I	5500 BCS Digimat 2000			I I
	Norwegian Telecom Administration	ND 100	SINTRAN III	I 8086	I	X25 Costal radio-control Experimental National N/& O Network Experimental service integrated video-switch			U U P P
Netherlands	Administration	PDP 10	(TOPS-10 (TOPS-20	PDP-11 MC 68000 PDP-10	I I U		Support system PRX/A OMC	OS Signalling C7 ISDN D-channel protocol Data base application Terminal multiplexes	I I U U U P I I
	AT&T and Philips Telecom	IBM 370	UNIX	IBM 370	U				

[1] P: Planned.
U: Under development.
I: Implemented.

Printed in Switzerland — ISBN 92-61-02251-0