# Reference Manual of the LNT to LOTOS Translator

*(Formerly: Reference Manual of the LOTOS NT to LOTOS Translator)*

## (Version 7.5)

David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Frédéric Lang,
Christine McKinty, Vincent Powazny, Wendelin Serwe, and Gideon Smeding

INRIA/VASY – INRIA/CONVECS

*December 17, 2024*

Note: this PDF document contains hyperlinks written in grey, whereas the normal text is in black.

# Abstract

This document defines the LNT language (version 7.5), which is a simplified variant of E-LOTOS (International Standard ISO-15437:2001). In a nutshell, LNT provides the same expressiveness as LOTOS, but has more user-friendly and regular notations borrowed from imperative and functional programming languages. In particular, unlike LOTOS, the data type and process parts of LNT share many similar constructs, leading to a more uniform and easy-to-learn language than LOTOS. This document defines the syntax, static semantics, and dynamic semantics of LNT, and presents its associated tools: the LNT2LOTOS translator and the LNT.OPEN script that interfaces with the OPEN/CÆSAR framework so as to enable LNT specifications to be analyzed using the CADP toolbox.

# Contents

# Chapter 1

# Introduction

## 1.1 Goals

This document defines the LNT language for specifying safety-critical systems.

### 1.1.1 A brief history of LOTOS and E-LOTOS

The LOTOS language [ISO89] was designed by experts in FDT (Formal Description Techniques) at ISO during the years 1981-1988. The objective was to design an *expressive*, *well-defined*, *well-structured*, and *abstract* language.

LOTOS has been used to describe numerous complex systems formally. A number of tools have been developed for LOTOS, covering user needs in the areas of simulation, compilation, test generation, and formal verification.

However, LOTOS actually has certain limitations, notably that the data types do not meet users' needs and the inability to specify real-time constraints.

For these reasons, ISO/IEC undertook in 1993 a revision of the LOTOS standard. This revision completed in 2001 with a new International Standard [ISO01]. The revised language is called E-LOTOS (for Extended-LOTOS). The enhancements of LOTOS are intended to remove known limitations of the language concerning expressiveness, abstraction and structuring capabilities, and user friendliness.

### 1.1.2 The LNT (formerly LOTOS NT) language

LNT is a language that follows the main concepts of E-LOTOS and offers other features, in order to provide versatility, as well as compilation and verification efficiency.

One major advantage of LNT is that its syntax is imperative, and thus easy to learn for developers and computer scientists.

Moreover, the purpose of LNT is to be both a concise language for small specifications (the so-called *programming in the small* level) and a suitable language for large specifications, with the ability to structure a project for team work (the so-called *programming in the large* level). While E-LOTOS is good only at the second point, LNT tries to address both needs.

The rationale for the semantic foundations of LNT are discussed in the four following publications:

[Gar95] (gate typing), [GS96] (exceptions), [GS99] (parallel composition), and [Gar15] (sequential composition).

So far, LNT has been been implemented in two different compilers, TRAIAN and LNT2LOTOS:

- TRAIAN[1] is a translator developed by the VASY and CONVECS teams since 1996. It takes as input a LNT specification and generates corresponding C code. The current version of TRAIAN only compiles the data part (type definitions and function definitions) of LNT. The LNT User Manual [SCC+24] describes the syntax and semantics of LNT, and lists the main differences between LNT and E-LOTOS.

- LNT2LOTOS is a translator from LNT to LOTOS, the development of which was undertaken in 2005, the VASY team undertook, as Bull's request. his translator enabled one to reuse the LOTOS-to-C compilers (namely, CÆSAR.ADT and CÆSAR) available in the CADP toolbox[2]. This translator progressively expanded in the framework of the FORMALFAME[3] and MULTIVAL[4] industrial projects.

When the development of LNT2LOTOS started, the initial goal was to reuse the same language as TRAIAN. However, while developing the tools and gaining industrial feedback from Bull, extensions (e.g., array types) and restrictions (related to translation issues) have been brought to the input language of LNT2LOTOS. This progressively led to two diverging versions of the language.

Between 2005 and 2014, the name "LOTOS NT" has been used for both languages supported by TRAIAN and LNT2LOTOS.

Between 2014 and October 2023, "LOTOS NT" was exclusively used to refer to the input language of TRAIAN, whereas the name "LNT" (a shorthand for LOTOS NT) became the official name to designate the input language accepted by LNT2LOTOS.

As of October 2023, after making a significant effort to reduce the differences between LOTOS NT and LNT, "LNT" is now used as the name for the common input language of TRAIAN and LNT2LOTOS.

A retrospective overview of the evolution of LOTOS and its descendents E-LOTOS, LOTOS NT, and LNT can be found in [GLS17].

### 1.1.3 LNT-to-LOTOS translation

This document describes the LNT language as accepted by the LNT-to-LOTOS translation tools LNT.OPEN and LNT2LOTOS.

The role of the LNT.OPEN and LNT2LOTOS tools that are presented in this document is to translate specifications written in the LNT language into LOTOS code that can be taken as input by the CADP tools.

## 1.2 Document structure

This document first explains how to use the translation tools LNT.OPEN and LNT2LOTOS to apply the CADP verification toolbox to LNT specifications (Chapter 2).

[1] http://vasy.inria.fr/traian
[2] http://cadp.inria.fr
[3] http://vasy.inria.fr/dyade/formalfame.html
[4] http://vasy.inria.fr/multival

Chapters 3 to 8 describe the syntax and semantics of the LNT language: its basic features (lexical structure, reserved keywords, etc.), the definition of modules (Chapter 4), the definition of data types (Chapter 5), the definition of functions (Chapter 7), and last, but not least, the definition of channels, behaviours, and processes (Chapter 8).

Appendix A contains a summary of the LNT syntax. by LNT2LOTOS.

Appendix B provides a formal semantics for LNT.

Appendix C contains a list of all the predefined functions.

A set of examples is given in Appendix D. They show how to define and use different kinds of LNT types, and explain how to use LNT types in LOTOS specifications, and LOTOS sorts in LNT programs. They also show how to define LNT functions.

Appendix E contains a summary of the current differences between LNT2LOTOS and TRAIAN.

Appendix F provides detailed examples showing how LNT constants are translated to LOTOS by LNT2LOTOS.

Appendix G gives the history of versions and changes for the LNT language and the associated tools.

# Chapter 2

# Overview of the translation from LNT to LOTOS

This chapter presents the translation of LNT into LOTOS and the related tools. For a detailed description of the tools, their options and usage, please refer to their manual pages.

## 2.1 Modules and principal module

A typical LNT specification consists of some LNT modules written in files with extension "`.lnt`". LNT modules can import other LNT modules, as explained in subsection 2.6. The module that transitively imports all other modules of the specification is called the *principal module*.

## 2.2 Root process

One of the modules must contain the *root process*, i.e., a process that is in general named "`MAIN`" unless the name of this process is specified on the command line using the "`-root`" option.

The root process is usually located in the principal module, but this is not mandatory.

## 2.3 Tools for translation of LNT into LOTOS

For details of how to use these tools, see their manual pages.

- LNT.OPEN is a script providing a connection between LNT2LOTOS and the OPEN/CÆSAR environment. The script automates the conversion of LNT programs to LOTOS code, by automatically calling TRAIAN, LNT2LOTOS, and finally CÆSAR.OPEN (which invokes CÆSAR.ADT and CÆSAR). See the LNT.OPEN manual page for details of its features, including, notably, multi-module compilation.

  LNT.OPEN takes as input the principal module of an LNT specification and an OPEN/CÆSAR application program. LNT.OPEN first translates the complete LNT specification (i.e., the principal module and all included modules) into LOTOS, compiles the generated LOTOS specification, and finally calls the OPEN/CÆSAR application program. Thus LNT.OPEN tries to automate and hide the translation steps as much as possible.

LNT.OPEN is the recommended tool for using LNT specifications in conjunction with CADP.

- TRAIAN is a compiler from LNT to C, whose front-end is used to perform advanced semantic analyses of the input LNT specification, providing useful error and warning messages. The C code generation is deactivated when TRAIAN is called from LNT.OPEN.

  The input file contains user-written LNT code.

  The output file contains the resulting code translated from the input file.

- LNT2LOTOS translates the LNT program into LOTOS.

  The input file must be a valid LNT program according to the specifications given in Chapters 3, 4, 5, 7, and 8.

  The output file contains the resulting LOTOS code translated from the input file.

## 2.4 File types and extensions

Each LNT module is translated into three output files:

- A LOTOS library (written in a file with extension "`.lib`") or, in case of the principal module, a LOTOS specification (written in a file with extension "`.lotos`")
- A "`.f`" file
- A "`.t`" file

The LNT.OPEN tool automates the translation of an LNT specification into LOTOS and the connection to the OPEN/CÆSAR interface of CADP.

An example of a project using the CADP verification tools to analyze a set of LNT modules is shown in Figure 2.1.

## 2.5 Including external C code

Optional external C code can be provided to LNT2LOTOS in a "`.fnt`" file for functions or a "`.tnt`" file for data type definitions (these files play for LNT2LOTOS the same role as the "`.f`" and "`.t`" files for CÆSAR and CÆSAR.ADT). The "`.fnt`" file must contain the line

```
#define LNT2LOTOS_EXPERT_FNT 7.5
```

The "`.tnt`" file must contain the line

```
#define LNT2LOTOS_EXPERT_TNT 7.5
```

These files are read by LNT2LOTOS and the contents are included in the generated "`.f`" or "`.t`" files. The names of all the files that define a module (the "`.lnt`" file and its optional "`.tnt`" and "`.fnt`" files) must be written in exactly the same way, including matching in case.

The version number tag, 7.5, is checked by LNT2LOTOS in the "`.fnt`" and "`.tnt`" files and by CÆSAR.ADT in the "`.f`" and "`.t`" files.

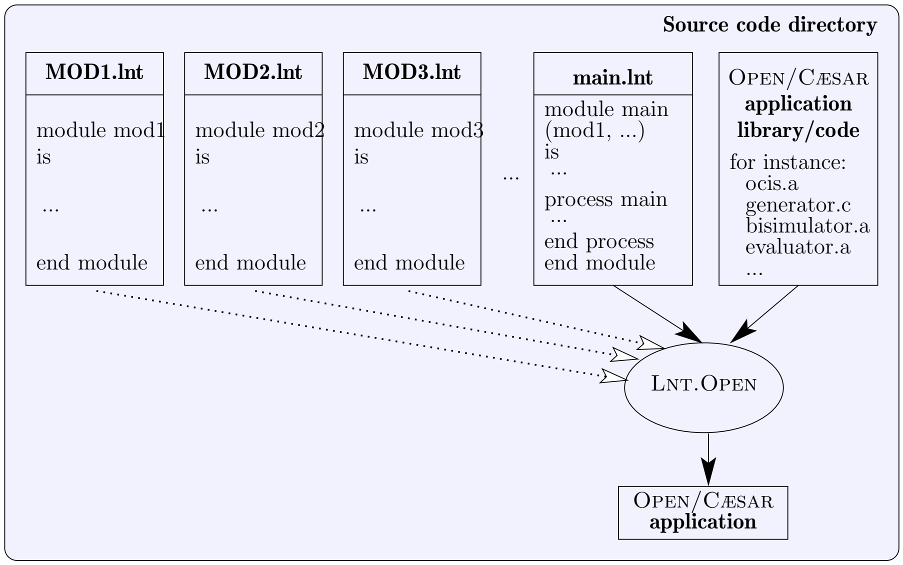


Figure 2.1: Using Lnt.Open to apply an Open/Cæsar application to an Lnt specification

## 2.6 Lnt modularity and file separation

Lnt is more modular than Lotos: each Lnt file contains exactly one module definition, and both the file and the module must have the same name. Letter case is not significant: a module `example` can indifferently be defined in a file named `Example.lnt`, `example.lnt` or `EXAMPLE.lnt`.

However, any import of the module must use the precise name of the file containing the module (respecting lower and upper case exactly).

A module `M` can import other modules `M0`, ..., `Mn` as follows:

```
module M (M0, ..., Mn) is
   -- module expression
   ...
end module
```

In such a module `M`, all definitions of `M0`, `M1`, ... and `Mn` are visible and can be used in the definitions in `M`.

Lnt2Lotos handles nested includes by importing all the modules directly into the auxiliary file of the principal module. For example, if principal module “`X.lnt`” directly imports modules “`Y.lnt`”, the auxiliary file “`X.f`” will include the file “`Y.f`”. The auxiliary “`.t`” file is generated using the same method as the “`.f`” file. It also automatically includes “`LNT_V1.h`”, so this must not be included in the hand-written “`.fnt`” or “`.tnt`” files. Lnt2Lotos detects and avoids multiple inclusions of the same code.

The included modules are searched first in the directory of the principal module, and then in the directory “`$LNT_LOCATION/lib`”. This allows the existence of a collection of predefined Lnt libraries. Current examples of such libraries can be found in “`$CADP/lib/BIT.lnt`” and “`$CADP/lib/OCTET.lnt`”.

In future releases, modules will import interfaces, which are the visible parts of modules. In the current release, no distinction is made between interfaces and modules: all definitions of a module (types, functions, channels, and processes) are considered visible.

## 2.7 Naming translation rules

Lnt2Lotos respects the following rules when translating Lnt to Lotos:

1. An Lnt **file** or **file.lnt** is translated into a Lotos **FILE** whose name is obtained by uppercasing the source file name and changing its extension to "`.lib`" (or "`.lotos`" for the principal module).
2. An Lnt **module** is translated into a Lotos **type** with the same name.
3. An Lnt **type** is translated into a Lotos **sort** with the same name.
4. An Lnt type **constructor** is translated into an Lnt constructor **operation** with the same name.
5. An Lnt **function** that returns a value and has neither "**out**" nor "**in out**" parameters, or that returns no value and has at most one "**out**" or "**in out**" parameter is translated into a Lotos function with the same name. Otherwise, the Lnt function is translated into several Lotos functions, whose names are unspecified.

Moreover, if a type or function name would result in a clash with a Lotos keyword, then the name is prefixed with respectively "`TYPE__`" or "`FUNC__`".

## 2.8 Environment variables

The `$LNT_LOCATION` shell environment variable should refer to the Lnt2Lotos installation directory. If this variable is not defined, the value of `$CADP` is used.

The environment variable `$PATH` should be modified in order to include the directories `$LNT_LOCATION/bin.‘$CADP/com/arch‘` and `$LNT_LOCATION/com`.

The files generated by Lnt2Lotos are stored in a separate directory, so that there is no confusion between the source code written by the user and the generated code.

The `$LNTGEN` environment variable should specify the path to this directory. Note that, if this path is relative to the source code directory, the same environment variable can be used for several projects.

If `$LNTGEN` is undefined in the current environment, "`./LNTGEN`" is used instead.

If the resulting path does not point to an existing directory, Lnt2Lotos tries to create it. If the creation fails, they issue an error message and stop.

## 2.9 Semantic checks

In general, the static semantic rules given in Chapters 4, 5, 6, 7, and 8 are checked at compile-time by Traian.

In few cases however, the checks are deferred to other tools and performed later. Such cases are indicated using the following notations:

- [checked by LNT2LOTOS] means that an error message can be raised at translation time by LNT2LOTOS.
- [checked by CÆSAR/CÆSAR.ADT] means that an error message can be raised by CÆSAR/CÆSAR.ADT when compiling the LOTOS code generated by LNT2LOTOS.
- [checked at runtime] means that an error message may be raised when the generated LOTOS code is executed.

# Chapter 3

# Notations and lexical elements

This chapter describes the lexical structure of the LNT language.

## 3.1 Meta-language

In this manual, to specify the concrete syntax of LNT, we use BNF (Backus-Naur Form) grammars extended with the following notations:

- $\varepsilon$ denotes the empty string
- [...] is the optional operator (0 or 1 instance)
- $y_0...y_n$ is the concatenation of one or more $y$ characters
- $y_1...y_n$ is the concatenation of zero or more $y$ characters
- $y_0, ..., y_n$ is the concatenation of one or more $y$ characters separated by commas
- $y_1, ..., y_n$ is the concatenation of zero or more $y$ characters separated by commas

## 3.2 Comments

In addition to LOTOS-like block comments of the form "`(* text *)`", single-line comments of the form "`-- text`" can be used in LNT. For the latter, all the text from the characters "`--`" to the end of the line is ignored.

LNT2LOTOS removes first the block comments, then the line comments, allowing line comments to be used within block comments.

## 3.3 Keywords

All LNT keywords must be written using lowercase letters. The list of LNT keywords is the following:

| | | | | | | |
|---|---|---|---|---|---|---|
| `access` | `alt` | `and` | `any` | `array` | `as` | `assert` |
| `break` | `by` | `case` | `channel` | `disrupt` | `div` | `else` |
| `elsif` | `end` | `ensure` | `eval` | `for` | `function` | `hide` |
| `if` | `in` | `is` | `list` | `loop` | `mod` | `module` |
| `null` | `of` | `only` | `or` | `out` | `par` | `process` |
| `raise` | `range` | `rem` | `require` | `result` | `return` | `select` |
| `set` | `sorted` | `stop` | `then` | `trap` | `type` | `use` |
| `var` | `where` | `while` | `with` | `xor` | | |

The identifiers of types, functions without "**out**" and "**in out**" parameters, processes, and gates present in the source LNT program are kept unchanged in the generated LOTOS program. Therefore, if such identifiers are LOTOS keywords, then LNT2LOTOS prints an error message rather than generating syntactically incorrect code. As a reminder, the list of LOTOS keywords is the following (those written in *italic font* are also keywords of LNT):

| | | | | | |
|---|---|---|---|---|---|
| `accept` | `actualizedby` | *any* | `behavior` | `behaviour` | `choice` |
| `endlib` | `endproc` | `endspec` | `endtype` | `eqns` | `exit` |
| *for* | `forall` | `formaleqns` | `formalopns` | `formalsorts` | *hide* |
| `i` | *in* | *is* | `let` | `library` | `noexit` |
| *of* | `ofsort` | `opnnames` | `opns` | *par* | *process* |
| `renamedby` | `sortnames` | `sorts` | `specification` | *stop* | *type* |
| `using` | *where* | | | | |

Note: In standard LOTOS, the token "`i`", which represents the internal gate (see Section 6.6), is a reserved keyword; it is thus impossible for the user to declare any identifier named "`i`", even if this identifier does not represent an event. In LNT, "`i`" is not a reserved keyword, but a predefined event identifier: it can thus be used without any restriction for naming LNT modules, types, constructors, channels, functions, variables, etc. However, "`i`" retains its special meaning when used as an event: thus, it is forbidden to declare an event named "`i`", to pass "`i`" as an actual event parameter in a process call, or to require synchronization on "`i`" in a parallel composition.

## 3.4 Identifiers

There are three types of identifiers:

- A "**normal-identifier**" consists of a letter optionally followed by any number of letters, digits, or underscores. It cannot start or end with an underscore, and cannot contain consecutive underscores. Examples of normal-identifier names are: "`Main`", "`timer_27`", "`x_6_p`".

- A "**special-identifier-1**" consists of a digit optionally followed by any number of letters or digits. Examples of special-identifier-1 names are: "`99catchall`", "`0start`".

  An identifier denoting a natural or integer constant, e.g., "123" or "0b11" (see Sections 3.5 and 3.6) is considered as such (rather than as a special-identifier-1).

- A "**special-identifier-2**" consists of a sequence of one or more of the following characters: "`#`", "`%`", "`&`", "`*`", "`+`", "`-`", "`/`", "`>`", "`=`", "`<`", "`@`", "", "`^`", "`~`". Examples of special-identifier-2 names are: "`>=`", "`<>`", "`**`".

  The sequence "`!=`" is also considered a special-identifier-2, even though the "`!`" is not normally permitted.

Identifiers are not case-sensitive. For instance, "my_idf", "My_Idf", and "MY_IDF" denote the same module, type, type constructor, function, variable, loop label, channel, event, or process. Note however that:

- Keywords are reserved, meaning that they cannot be used as identifiers, except "`and`", "`div`", "`mod`", "`or`", "`rem`", and "`xor`", which can be used as constructor or function identifiers.
- Identifiers obtained by turning some lowercase letter to uppercase are valid identifiers. For instance, "END" and "Var" are valid identifiers whereas "`end`" and "`var`" are keywords.

As a general rule, when using a module, a type, a type constructor, a function, a variable, a loop label, a channel, an event, or a process identifier, it is recommended to use the same letter case as its definition whenever possible.

Constructor and function identifiers can be any identifier-type, whereas other identifiers are normal-identifier type, as shown below:

| Identifier | Meaning | Identifier type |
|---|---|---|
| $M$ | *module* | normal-identifier |
| $T$ | *type* | normal-identifier |
| $C$ | *type constructor* | normal-identifier or special-identifier-1 or special-identifier-2 |
| $X$ | *variable* | normal-identifier |
| $F$ | *function* | normal-identifier or special-identifier-1 or special-identifier-2 |
| $L$ | *loop label* | normal-identifier |
| $\Gamma$ | *channel* | normal-identifier |
| $E$ | *event* | normal-identifier |
| $\Pi$ | *process* | normal-identifier |

The use of LOTOS keywords as identifiers should be avoided (see Section 3.3 above).

## 3.5 Natural numbers

With LNT, natural number notations can be used as in any programming language. The notations are those of the Microsoft's F# language. They were preferred to those of C, C++ and JAVA. Firstly, these last three languages lack a notation for binary numbers. Secondly, there is a risk of confusion between decimal and octal notations: a number notation which only contains digits can either be decimal (`756`) or octal (`0756`).

LNT supports four notations:

| | | | |
|---|---|---|---|
| *bindigit* | ::= | `0`\|`1` | *binary digit* |
| *octdigit* | ::= | `0`\|`1`\|`2`\|`3`\|`4`\|`5`\|`6`\|`7` | *octal digit* |
| *decdigit* | ::= | `0`\|`1`\|`2`\|`3`\|`4`\|`5`\|`6`\|`7`\|`8`\|`9` | *decimal digit* |
| *hexdigit* | ::= | `0`\|`1`\|`2`\|`3`\|`4`\|`5`\|`6`\|`7`\|`8`\|`9`\|`a`\|`A`\|`b`\|`B`\|`c`\|`C`\|`d`\|`D`\|`e`\|`E`\|`f`\|`F` | *hexadecimal digit* |
| *nat* | ::= | *decdigit*+ | *decimal constant, e.g.,* `34` |
| | \| | `0x`*hexdigit*+ | *hexadecimal constant e.g.,* `0xf2` |
| | \| | `0o`*octdigit*+ | *octal constant e.g.,* `0o42` |
| | \| | `0b`*bindigit*+ | *binary constant e.g.,* `0b10010` |

In addition, leading zeros are forbidden in decimal numbers, so that the only syntactically valid decimal number starting with a zero is 0.

For readability, the "`_`" character can be used to separate groups of digits, as in Ada or VHDL; it is just a convenient syntactic notation for writing numbers, without semantic meaning. The "`_`" character is accepted anywhere in natural number notations except before the first digit or after the last digit. Consecutive "`_`" characters are not allowed. Some examples of correct expressions are: `19_785`, `0xAFF_BCDE`, `0o3_377`, `0b110_0110_0111`.

By default, natural numbers in LNT are assumed to be in the range 0..255. This is explained by the fact one wants to avoid large numbers that increase complexity in explicit state model checking. If an LNT specification handles larger numbers than 255, an overflow error is likely to occur at run-time. However, the domain of natural numbers can easily be enlarged using either the "`!nat_bits`", the "`!num_bits`", the "`!num_card`", or the "`!nat_inf/!nat_sup`" pragmas (see Section 4.4 for details).

## 3.6 Integer numbers

Integer numbers can be either positive, negative or zero.

By default, integer numbers in LNT are assumed to be in the range −128..127. This is explained by the fact one wants to avoid large numbers that increase complexity in explicit state model checking. If an LNT specification handles larger numbers, an underflow or overflow error is likely to occur at run-time. However, the domain of integer numbers can easily be enlarged using either the "`!int_bits`" or the "`!int_inf/!int_sup`" pragmas (see Section 4.4 for details).

All the notations available for natural numbers are also available for integer numbers. Here are some examples of integer numbers: `0`, `123`, `-123`, `0x4`, `-0xFD`, `-0o76`, `-0b1011`, etc.

Explicit type casts can be used to resolve typing ambiguities that may arise between natural numbers and integer numbers: for instance, one can distinguish between `12 of Nat` and `12 of Int`. Note that explicit type casts "`of Int`" are superfluous for integer number with a unary operation "`+`" or "`-`".

As with natural numbers, the "`_`" character can be used to separate groups of digits.

Note: integer numbers preceded by a unary "`-`" without parentheses are considered as negative integer constants rather than applications of the unary operator "`-`" to a positive integer constant. This has the advantage of allowing to write the constant $-2^{k-1}$, even when integers are represented using $k$ bits. Notice that writing "$-(2^{k-1})$" yields an integer overflow, because "$2^{k-1}$" is not an admissible integer value when integers are represented using $k$ bits.

## 3.7 Real numbers

Reals (i.e., floating-point numbers) can be written as in classical programming languages. The LNT syntax is inspired from the floating-point numbers of the C programming language, with a few restrictions and an extension.

A floating-point number is a non-empty sequence of digits optionally containing a decimal point, followed by an optional exponent part. At least one of the two optional parts (decimal point or exponent) must be present. If the decimal point is present, it must be preceded and followed by at least one digit. A floating-point number cannot have a leading zero unless the zero is immediately followed by the decimal point or by the exponent. Leading zeros are accepted in the exponent.

As natural in numbers, the "`_`" character can be used to separate groups of digits; The "`_`" character is accepted anywhere in between two digits. Consecutive "`_`" characters are not allowed.

Here are some examples of floating-point numbers: `0.1`, `0.2`, `3.0e-1`, `7.49E-005`, `5_521.49_61E-0_0_5`, `4.0e0`, `5.0`, `0E0`, etc.

The following notations (some of which are available in the C programming language) are not available in LNT: `3._14`, `0_.1`, `.1`, `02.87e-10`, `3.e-1`, `5.`, `7.4_`, `00E0`, `_6.21`, etc.

## 3.8 Characters

Characters of type `Char` are C-like (unsigned) characters enclosed into single quotes:

- Any ASCII character: "`a`", "`é`", "`|`", "`0`", etc.
- C escape sequence shortcuts for non-printing characters (carriage return, tabulation, etc.):

| ASCII Name | Description | C escape sequence |
|---|---|---|
| `nul` | null byte | `\0` |
| `bel` | bell character | `\a` |
| `bs` | backspace | `\b` |
| `np` | formfeed | `\f` |
| `nl` | newline | `\n` |
| `cr` | carriage return | `\r` |
| `ht` | horizontal tab | `\t` |
| `vt` | vertical tab | `\v` |

- C standard escape sequences:

| Printable character | C escape sequence |
|---|---|
| `"` | `\"` |
| `\` | `\\` |
| `'` | `\'` |
| `?` | `\?` |

- Restricted C-like octal or hexadecimal escape sequences:
  - An octal escape sequence `\`$ooo$ with exactly three octal digits $o$ ($o \in [0..7]$) where the $ooo$ octal value is less than or equal to `\377`. Escape sequences with less than or more than three digits, like `\1`, `\01` or `\0001`, are rejected.
  - A hexadecimal escape sequence `\x`$hh$ with exactly two hexadecimal digits $h$ ($h \in [0..9, A..F]$). Escape sequences with less than or more than two digits, like `\x1` or `\x001`, are rejected.

Note: The same character can be written using different notations in LNT. For instance, the null character can be written either "`\0`", or "`\000`", or "`\x00`"; the newline character can be written either "`\n`", or "`\12`", or "`\012`"; and so on.

Character values will be displayed surrounded by single quotes. All printable characters will be displayed as such, e.g., `'a'`, `'b'`, `'c'`, etc. All non-printable characters (e.g., control characters) will be displayed using three-byte octal notation `'\`$ooo$`'`, where $o$ is an octal digit. The single quote and backslash characters are displayed as `'\''` and `'\\'`, respectively.

## 3.9 Strings

The `String` constants are C-like strings: they consist of character sequences enclosed in double quotes. The characters supported are the same as for the `Char` type. There can also be any ASCII character, C escape sequence shortcuts for non-printing characters (carriage return, tabulation, etc.), C standard escape sequences, and restricted C-like octal or hexadecimal escape sequences, for example:

```
""
"éêè"
"2\nlines"
"\""
"’"
"\’"
"\\"
"AZERTY"
"A\x5AERTY"
"A\132ERTY"
```

String values will be displayed surrounded by double quotes. All printable characters will be displayed as such in strings, e.g., `"...abc..."`. All non-printable characters (e.g., control characters) will be displayed using three-byte octal notation `"...\`*ooo*`..."`, where *o* is an octal digit. The double quote and backslash characters are displayed as `"...\"..."` and `"...\\..."`, respectively.

## 3.10 Prefix and infix calls of constructors and functions

In general, a call to any constructor or function identifier, whatever its number of arguments $(0, 1, 2, ...n)$, can be done in prefix notation followed by parentheses: e.g. “`@()`”, “`f (1)`”, “`g (x, y)`”, or “`+ (1, 2)`”.

For convenience, the following extensions are provided:

- A call to a nullary constructor or function identifier that is a normal-identifier or special-identifier-1 can be done without parentheses. For example: “`pi`” instead of “`pi()`”, “`2F3A`” instead of “`2F3A()`”.

  Note: A nullary that is special-identifier-2 must be called with parentheses, e.g., “`@()`” instead of “`@`”.

  Note: If a variable and a constructor or function have the same names, putting parentheses after the constructor avoids ambiguity and distinguishes between them. If no parentheses are used, the variable masks the constructor.

- A call to a unary constructor or function identifier that is a special-identifier-2 can be done without parentheses: e.g. “`-`$n$” or “`-(`$n$`)`” are syntactically correct, but are semantically different if $n = 2^k$ and integer numbers are represented using $k$ bits, as explained in Section 3.6 for details.

  Note: Sequences of unary constructors or functions without parentheses are not recommended, because the proper handling of negative integer numbers is only ensured for sequences of *odd* length. For instance, if $k$ bits are used to represent values of type `Int` and function `@` is defined as

  ```
  function @ (X: Int) : Int is
  ```

```
   return (X % 2)
end function
```

“`@ -2`$^{k-1}$” yields an overflow, whereas “`@(-2`$^{k-1}$`)`” does not.

Note: A call to a unary constructor or function identifier that is a normal-identifier or special-identifier-1 must be done with parentheses: e.g. “`f (x)`” and not “`f x`”.

- A call to a binary operator that is a normal-identifier or special-identifier-2 can also be done in an infixed way, e.g. :“ `1 div 2`”, “`1 mod 2`”, “`1 + 2`” (in addition to “`div (1, 2)`”, “`mod (1, 2)`”, and “`+ (1, 2)`”).

  Note: A binary operator that is a special-identifier-1 must be used in prefix mode, i.e., “`000 (x, y)`” and not “`x 000 y`”.

# Chapter 4

# Module definitions in LNT

## 4.1 Notations

This chapter uses the BNF notations defined in Section 3.1.

The following additional convention is used:

- $M$ is a module identifier

## 4.2 Syntax

| | | | |
|---|---|---|---|
| *lnt_file* | ::= | **module** $M$[($M_0, ..., M_m$)] | *module* |
| | | [**with** *predefined_function*$_0$, ..., *predefined_function*$_n$] **is** | |
| | | *module_pragma*$_1$...*module_pragma*$_p$ | |
| | | *definition*$_0$...*definition*$_q$ | |
| | | **end module** | |

| | | | |
|---|---|---|---|
| *predefined_function* | ::= | == \| = | *equality* |
| | \| | <> \| != | *inequality* |
| | \| | < | *less than* |
| | \| | <= | *less than or equal to* |
| | \| | > | *greater than* |
| | \| | >= | *greater than or equal to* |
| | \| | append | *tail insertion* |
| | \| | card | *set cardinality* |
| | \| | delete | *element deletion* |
| | \| | diff | *asymmetric difference* |
| | \| | element | *indexed access* |

| | | | |
|---|---|---|---|
| | \| | `empty` | *emptyness test* |
| | \| | `first` | *first element* |
| | \| | `get` | *field selection* |
| | \| | `head` | *first element* |
| | \| | `insert` | *insertion* |
| | \| | `inter` | *intersection* |
| | \| | `last` | *last element* |
| | \| | `length` | *list length* |
| | \| | `member` | *membership test* |
| | \| | `minus` | *symmetric difference* |
| | \| | `ord` | *ordinal* |
| | \| | `remove` | *element removal* |
| | \| | `reverse` | *reversal* |
| | \| | `set` | *field update* |
| | \| | `subset` | *subset test* |
| | \| | `tail` | *next elements* |
| | \| | `union` | *union* |
| | \| | `val` | *value* |

| | | | |
|---|---|---|---|
| *module_pragma* | ::= | `!nat_bits` *nat* | *number of bits for type Nat* |
| | \| | `!nat_inf` *nat* | *lowest value of type Nat* |
| | \| | `!nat_sup` *nat* | *highest value of type Nat* |
| | \| | `!nat_check` *bit* | *check for Nat overflows/underflows* |
| | \| | `!int_bits` *nat* | *number of bits for type Int* |
| | \| | `!int_inf` *int* | *lowest value of type Int* |
| | \| | `!int_sup` *int* | *highest value of type Int* |
| | \| | `!int_check` *bit* | *check for Int overflows/underflows* |
| | \| | `!num_bits` *nat* | *number of bits for numeral types* |
| | \| | `!num_card` *nat* | *maximal cardinality for numeral types* |
| | \| | `!string_card` *nat* | *maximal cardinality for type String* |
| | \| | `!update` *string* | *update tag* |
| | \| | `!version` *string* | *version tag* |

where *nat* denotes a natural number constant (in decimal notation without underscores), *int* denotes an integer number constant (in decimal notation without underscores), and *bit* denotes 0 or 1.

| | | | |
|---|---|---|---|
| *definition* | ::= | *type_definition* | *type definition* |
| | \| | *function_definition* | *function definition* |
| | \| | *channel_definition* | *channel definition* |

| *process_definition* *process definition*

Type definitions are covered in Chapter 5. Chapter 7 describes function definitions. Process definitions are discussed in Chapter 8.

## 4.3 Module definitions

(MD1) The name of each file containing an LNT module must have the "`.lnt`" extension. The characters used in such a file name can only be letters, digits, and underscore ("_") — in addition to the dot occurring in the extension.

(MD2) Module $M$ must have the same name as the file in which it is defined (without extension). Letter case is significant. For instance, a module "MyModule" has to be defined in a file named *MyModule*`.lnt`. Other names such as *mymodule*`.lnt` or *MYMODULE*`.lnt` are not allowed.

However, in the particular case where the module name is "TEST" (or "Test", "test", etc.), having a different file name only triggers a warning, whereas in all other cases a fatal error is issued if the module name and file name do not match. This particular case is intended to ease debugging and rapid prototyping.

(MD3) The identifiers $M_0, ..., M_m$ must refer to different modules, and must be different from $M$.

(MD4) The "**`with`**" clause of a module requests the corresponding predefined functions to be present for all the types declared inside the module (if these functions exist for these types). Each predefined function must be declared only once in the "**`with`**" clause of a given module.

Additional information about the semantics of predefined functions can be found in Section 5.7.

## 4.4 Module pragmas

Module pragmas can be used to modify the default settings related to the implementation of the predefined types `Nat`, `Int`, and `String`.

(MP1) All module pragmas but `!version` and `!update` must only appear in the principal module (see Section 2.1). Otherwise a warning will be emitted. [checked by LNT2LOTOS]

(MP2) Each pragma but `!version` and `!update` must appear at most once. [checked by LNT2LOTOS]

(MP3) Pragma `!update` must appear at most once in each module. [checked by LNT2LOTOS]

(MP4) The module pragmas "`!num_bits`" and "`!num_card`" are mutually exclusive.

(MP5) The value *nat* of the pragmas "`!num_bits`" and "`!num_card`" must be natural numbers.

(MP6) The value *nat* of a pragma "`!num_bits`" should be different from 0; even if value zero might be tolerated in some cases, its precise effect is undocumented.

(MP7) The value *nat* of a pragma "`!num_card`" should be different from 0 and 1; even if these two values are tolerated in some cases, their precise effect is undocumented. [checked by LNT2LOTOS]

The module pragmas have the following effect:

- `!nat_bits` $N$ (resp. `!int_bits` $N$) specifies the number of bits ($N > 0$) with which a value of type `Nat` (resp. `Int`) will be represented. By default, $N = 8$.
- `!nat_inf` $N'$ and `!nat_sup` $N''$ respectively denote the lowest and highest values to be used when iterating on the `Nat` domain. By default, $N' = 0$ and $N'' = 2^N - 1$, where $N$ is the number of bits for type `Nat`.
- `!int_inf` $I'$ and `!int_sup` $I''$ respectively denote the lowest and highest values to be used when iterating on the `Int` domain. By default, $I' = -2^{N-1}$ and $I'' = 2^{N-1} - 1$, where $N$ is the number of bits for type `Int`.
- `!nat_check` $B$ (resp. `!int_check` $B$) specifies whether numeric overflows/underflows have to be checked ($B = 1$ means checked, $B = 0$ means unchecked) for type `Nat` (resp. *Int*). By default, $B = 1$, meaning that checks must be disabled explictly by setting $B = 0$.

  Note: The implementation of predefined libraries (in `$CADP/incl/X_NATURAL.h` and `$CADP/incl/X_INTEGER.h`) does its best to detect overflows and underflows, especially by performing computations on naturals/integers that are twice as large (in number of bits) than what is needed to store values of types `Nat` and `Int`. This can be slightly more CPU-intensive, but this is probably the price to pay for gaining increased confidence in an LNT specification. However, because of the limitations of the C language, some overflows or underflows may remain detected (e.g., if types `Nat` and `Int` have the maximum number of bits allowed on the machine, or if involved arithmetic operations are used).
- `!string_card` $N$ stores all character strings in a hash table with $N$ entries at most. Technically, this is achieved by setting the macro "`CAESAR_ADT_HASH_ADT_STRING`" to $N$ in the C code generated by LNT2LOTOS.
- `!update` can be used to declare that the module takes into account updates introduced in LNT up to the given CADP version.

  Currently, "`!update "2021-b"`" is the only update tag allowed. It means that the module takes into account the priorities of infix operators introduced in CADP 2021-b (February 2021) and described in Section 7.3. It allows to avoid warnings indicating that expressions are parsed differently due to this change.

  Beware that due to a current limitation of modules, the scope of update tags extends to all modules of the LNT program.

  Another way to avoid these warnings is to set the environment variable "`$LNT_UPDATE`" with the value `2021-b`.
- `!version` can be used to label the module with a version tag. This pragma has no effect in the generated code so far.
- "`!num_bits` $N$" specifies a maximal value $2^N$ for the number of elements of all *numeral* types $T$, meaning that each of these elements will be implemented in C using at most $N$ bits. A numeral type is any type isomorphic to "**type** $T$ **is** *Zero* **,** *Succ* **(**$X$**:**$T$**) end type**", where $T$, *Zero*, *Succ*, and $X$ can be arbitrary identifiers (the **nat** type is a particular case of numeral type).

By default, numeral types are implemented by LNT2LOTOS and CÆSAR.ADT using one single byte; the "`!num_bits`" pragma can be used to change this default setting to a number of bits different from eight.

The module pragmas "`!nat_bits`" and "`!nat_sup`", and the type pragmas "`!bits`" and "`!card`" (see Section 5.6) have precedence over "`!num_bits`". This means that, if both pragmas "`!num_bits`" and "`!nat_bits`" (or "`!nat_sup`") are present, natural numbers are encoded using the number of bits specified by "`!nat_bits`" (or "`!nat_sup`"), while the elements of any other numeral type $T$ are encoded using the number of bits specified by "`!num_bits`", unless the definition of $T$ has a "`!bits`" (or "`!card`") pragma, in which case the number of bits specified by this type pragma will be used for implementing $T$.

Note: This pragma is implemented by inserting a macro "`ADT_PRAGMA_NUMERAL -`$N$" in the generated C code.

- '`!num_card` $N$" specifies a maximal value $N$ for the number of elements of all numeral types (see previous item), whose values are thus in the range $0, ..., N-1$. The module pragmas "`!nat_bits`" and "`!nat_sup`", and the type pragmas "`!bits`" and "`!card`" have precedence over "`!num_card`".

  Note: This pragma is implemented by inserting a macro "`ADT_PRAGMA_NUMERAL` $N$" in the generated C code.

## 4.5 Constructors, functions, procedures, and processes

The LNT language has four different kinds of routines:

- A "*constructor*" (see Chapter 5) is a routine that has zero, one, or more arguments and that returns a single result. A constructor has only formal parameters of mode "**in**". A constructor is defined as part of the definition of the type of its result. The body of a constructor is never defined explicitly.
- A "*function*"[1] (see Chapter 7) is a routine that has zero, one, or more arguments and that returns a single result. A function has only formal parameters of mode "**in**" and/or "**in var**". Functions can be predefined, externally defined (i.e., written in LOTOS or in C), or defined by the user in LNT. The body of a user-defined function is an LNT statement (see Section 7.2), the simplest form being a "**return**" statement.
- A "*procedure*" (see Chapter 7) is a routine that has zero, one, or more arguments and that may return a result. A procedure can have formal parameters of mode "**in**", "**in var**", "**out**", "**out var**", or "**in out**". Procedures can be externally defined (i.e., written in C) or defined by the user in LNT. The body of a user-defined procedure is an LNT statement (see Section 7.2), which may or not contain a "**return**" statement.
- A "*process*" (see Chapter 8) is a routine that resembles a procedure, but has a greater expressiveness, as it can perform actions (i.e., inputs, outputs, communications, synchronizations, internal actions, etc.), nondeterministic choices, parallel composition, etc. Processes can be externally defined (i.e., written in LOTOS) or defined by the user in LNT. The body of a user-defined process is an LNT behaviour (see Section 8.2). Unlike processes, functions and procedures do not perform actions; they are deterministic and atomic (i.e., they execute in zero time). Conversely, a process does not return a result (i.e., it has no "**return**" statement).

[1]Functions are sometimes referred to as non-constructors.

Contrary to ALGOL-like languages (including Pascal, Ada) and like C-like languages (including C++ and Java), LNT does not make a syntactic distinction between functions and procedures. Both are declared using the same keyword "**function**" and, sometimes, the word *function* is used to designate either a function or a procedure. However, there are semantic differences between functions and procedures; for instance, only functions (but not procedures) can be used in expressions.

# Chapter 5

# Type definitions in LNT

## 5.1 Notations

This chapter uses the BNF notations defined in Section 3.1.
The following additional conventions are used:

- $T$ is a type identifier
- $C$ is a type constructor identifier
- $X$ is a variable identifier
- $V$ is a value expression (see Section 7.13)
- $m$ and $n$ are integer numbers in decimal notation without underscores ("_").

## 5.2 Syntax

| | | | |
|---|---|---|---|
| *type_definition* | ::= | **type** $T$ **is** *type_pragma*$_1$...*type_pragma*$_n$ | *type* |
| | | *type_expression* | |
| | | [**with** *predefined_function_declaration*$_0$, ..., *predefined_function_declaration*$_m$] | |
| | | **end type** | |

| | | | |
|---|---|---|---|
| *type_expression* | ::= | *constructor_definition*$_0$, ..., *constructor_definition*$_n$ | *constructed type* |
| | \| | **set of** $T$ | *set* |
| | \| | **list of** $T$ | *list* |
| | \| | **sorted list of** $T$ | *sorted list* |
| | \| | **array** $[m\,..\,n]$ **of** $T$ | *array* |
| | \| | **range** $m\,..\,n$ **of** $T'$ | *range* |

| | | |
|---|---|---|
| \| | $X : T'$ [**where** $V$] | *predicate* |
| \| | | *empty (external type only)* |

*constructor_definition* ::= $C$ [`(`*constructor_parameters*$_1$, ..., *constructor_parameters*$_n$`)`]
*constructor_pragma*$_1$...*constructor_pragma*$_m$

| | | | |
|---|---|---|---|
| *type_pragma* | ::= | `!external` | *external type* |
| | \| | `!implementedby "`[`C:`]*name*`"` | *C type name* |
| | \| | `!comparedby "`[`C:`]*name*`"` | *C equality function* |
| | \| | `!printedby "`[`C:`]*name*`"` | *C printing function* |
| | \| | `!list` | *print as list* |
| | \| | `!iteratedby "`[`C:`]*name*$_1$`" , "`[`C:`]*name*$_2$`"` | *C iterator functions* |
| | \| | `!pointer` | *C pointer implementation* |
| | \| | `!nopointer` | *C unboxed implementation* |
| | \| | `!bits` *nat* | *number of bits for the C type* |
| | \| | `!card` *nat* | *maximal cardinality for the C type* |

*predefined_function_declaration* ::= *predefined_function*
[*predefined_function_pragma*$_1$...*predefined_function_pragma*$_n$]

where *predefined_function* is defined in Section 4.2.

| | | | |
|---|---|---|---|
| *predefined_function_pragma* | ::= | `!external` | *external function* |
| | | `!implementedby "`[(`C` \| `LOTOS`)`:`]*name*`"` | *C/*LOTOS *name scheme* |

*constructor_parameters* ::= $X_0, ..., X_n : T$ *constructor parameters*

*constructor_pragma* ::= `!implementedby "`[`C:`]*name*`"` *C operator name*

## 5.3 Type definitions

(TD1) All types $T$ defined in module $M$ must have different identifiers.

(TD2) All types $T$ defined in module $M$ and in the imported modules $M_0, ..., M_n$ must have different identifiers.

(TD3) In the list $type\_pragma_1 ... type\_pragma_n$ of each type definition, there should be at most one pragma of each kind (i.e., there cannot be two "`!external`" pragmas, nor two "`!implementedby "..."`"/"`!implementedby "C:..."`" pragmas, etc.)

(TD4) Each predefined function must be declared only once in the "`with`" clause of a given type.

## 5.4 Type expressions

(TE1) $T$ must be the identifier of a type defined in the current module or in an imported module.

(TE2) When a type expression defines a **sorted list of** $T$ or a **set of** $T$, a comparison operator `<` must be defined for type $T$ and for all the types that are used to define $T$. Such an operator is automatically generated by LNT2LOTOS when clause `with <` is specified.

(TE3) When a comparison operator is requested (using a `with` clause) for a type $T$, a comparison operator `<` must be defined for each type appearing in the definition of $T$.

(TE4) When a type expression defines an **array**, the bounds $m$ and $n$ must be natural numbers such that $m \leq n$.

(TE5) When a type expression defines a **range**, the type $T'$ must be **Char**, **Int**, or **Nat**.

(TE6) When a type expression defines a range of **Char**, the bounds $m$ and $n$ must be character constants such that $m <= n$. In this case, $m$ and $n$ are expressed using the ASCII code of the characters.

(TE7) When a type expression defines a range of **Int**, the bounds $m$ and $n$ must be integer numbers such that $m <= n$.

(TE8) When a type expression defines a range of **Nat**, the bounds $m$ and $n$ must be natural numbers such that $m <= n$.

(TE9) A type definition using **set**, **list**, or **sorted list** should not be directly recursive. For instance, it is forbidden to write "**type** $T$ **is list of** $T$ **end type**" (such a definition is misleading, since $T$ does not correspond to a list, but to a binary tree). Notice that indirect (i.e., transitive) recursion by means of one or more auxiliary types is allowed. [checked by CÆSAR/CÆSAR.ADT, which emits warnings about incorrect "`!list`" pragmas (such pragmas are automatically added, but the type constructors do not have the right profiles)]

(TE10) A type expression can be empty only if the pragma "`!external`" is present.

The following array type definition:

**type** $T$ **is**
  **array** $[m\,..\,n]$ **of** $T'$
**end type**

is equivalent to defining a type $T$ with one constructor $T$ and $n - m + 1$ parameters of type $T'$.

Note: Array bounds are required to be natural numbers. This implies that they must be representable using 32 bits.

Note: LNT2LOTOS allocates memory for creating the LOTOS files. Defining a large array can lead to errors if there is insufficient memory for compilation. For example, an LNT specification containing the definition:

**type** $T$ **is**
  **array** [1..1000000000] **of** $Int$
**end type**

when compiled, may cause a stack overflow.

To initialize variables of type $T$, LNT2LOTOS provides a more convenient way than calling constructor $T$ with $n - m + 1$ parameters. Constructor $T$ is overloaded with an operation $T$ which takes one parameter $V$ of type $T'$ and builds an array that contains $n - m + 1$ times the same value $V$.

Moreover, the syntax defined in chapter 7 allows one to assign a value to an array element, and to use an array element in an expression.

A range type must be written with spaces before and after $m$ and $n$. For example, a definition containing

```
range -3..-2 of Int
```

will be rejected with an error message. It should instead be written as

```
range -3 .. -2 of Int
```

## 5.5 Constructor definitions

Note: Each list of constructor parameters "$X_0, ..., X_n : T$" is flattened into a list "$X_0 : T$, ..., $X_n : T$".

(CD1) Two or more constructors may have the same name (may be overloaded) if their profiles (the list of the types of fields) differ.

(CD2) All the constructor parameters $X_0, ..., X_n$ must have different identifiers.

(CD3) For the set of constructors of a given type, fields having the same name should have the same type.

(CD4) Each type $T_0, ..., T_n$ must refer to a type defined in the current module or in an imported module.

(CD5) In the list *constructor_pragma*$_1$...*constructor_pragma*$_n$ of each constructor definition, there should be at most one pragma of each kind (i.e., there cannot be two "`!implementedby "..."`"/"`!implementedby "C:..."`" pragmas).

(CD6) Type declarations may be mutually recursive. However, each type must be productive, *i.e.* it must have at least one value. Formally, a type is productive if and only if: (a) it has a constructor with arity 0 or (b) all the parameters of its constructors have productive types.

(CD7) A constructor $C$ with two parameters can be used both in infix or prefix forms (i.e., both "$x\ C\ y$" or "$C(x, y)$").

## 5.6 Type pragmas and constructor pragmas

(TCP1) In pragmas of the form "`!implementedby "`[`C:`]*name*`"`", "`!comparedby "`[`C:`]*name*`"`", and "`!printedby "`[`C:`]*name*`"`", *name* must be valid C function identifier. It must neither be a reserved keyword of the C language nor an identifier predefined in the standard libraries of the C language (e.g., "true", "false", "bool").

(TCP2) In pragmas of the form "`!iteratedby "`[`C:`]$name_1$`", "`[`C:`]$name_2$`"`", $name_1$ and $name_2$ must be valid C macro identifiers corresponding to CÆSAR.ADT iteration macros. They must neither be reserved keywords of the C language nor identifiers predefined in the standard libraries of the C language (e.g., "true", "false", "bool").

(TCP3) The pragma "`!external`" must not be given for set, list, sorted list, array, range, or predicate types. Otherwise, a warning message is issued and the pragma is ignored.

(TCP4) The pragma "`!list`" should be given only to a type $T$ having a list-like structure, i.e., $T$ should have exactly two constructors, a first one, usually called "`nil`", without parameters and a second one, usually called "`cons`", with two parameters, exactly one of which is of type $T$ (see Section 5.7).

(TCP5) For **list**, **sorted list**, and **set** types, the type pragma `!list` is automatically added if it is not specified already.

(TCP6) The type pragmas "`!pointer`","`!nopointer`", "`!bits`", and "`!card`" are mutually exclusive.

(TCP7) The value *nat* of the pragmas "`!bits`" and "`!card`" must be natural numbers.

(TCP8) The value *nat* of a pragma "`!bits`" should be different from 0.

(TCP9) The value *nat* of a pragma "`!card`" should be different from 0 and 1. [checked by LNT2LOTOS]

(TCP10) The type pragmas "`!pointer`","`!nopointer`", "`!bits`", and "`!card`" should not be given for enumerated types (including singleton types, which are enumerated types with a single value). [checked by CÆSAR/CÆSAR.ADT]

(TCP11) The type pragmas "`!pointer`" and "`!nopointer`" should not be given for numeral types, i.e., types that have two constructors, one having no field and the other one having a single field of this same type (recursively). [checked by CÆSAR.ADT]

The pragmas attached to types and/or constructors have the following effects:

- The pragma "`!external`" indicates that the type (respectively, constructor) is defined by an external C type (respectively, C function); this pragma is translated into a special comment in the generated LOTOS code. For a type declared "`!external`", LNT2LOTOS automatically associates the "`!external`" pragma to all its constructors.

- If a type has the pragma "`!external`", then all constructors of this type (if any) and all functions declared in the "**with**" clause of this type (if any) are also external, i.e., an external definition of these constructors and functions must be provided in C code.

- The pragma "`!implementedby "C:`*name*`"`" (or simply "`!implementedby "`*name*`"`") indicates that the type (respectively, constructor) should be implemented by the C type (respectively, C function) named *name*; this pragma is translated into a special comment in the generated LOTOS code.
- The pragma "`!comparedby "C:`*name*`"`" (or simply "`!comparedby "`*name*`"`") indicates that the C function implementing the comparison of two elements of the type should be named *name*; this pragma is translated into a special comment in the generated LOTOS code.
- The pragma "`!printedby "C:`*name*`"`" (or simply "`!printedby "`*name*`"`") indicates that the C function printing elements of the type should be named *name*; this pragma is translated into a special comment in the generated LOTOS code.
- The pragma "`!list`" indicates that the type should be printed as a list, i.e., using the notation "$\{V_1, ..., V_n\}$"; this pragma is translated into a special comment in the generated LOTOS code.
- The pragma "`!iteratedby "C:`$name_1$`" , "C:`$name_2$`"`" (or simply "`!iteratedby "`$name_1$`" , "`$name_2$`"`") indicates that the two C macros implementing the iterator for the type should be named $name_1$ and $name_2$; this pragma is translated into a special comment in the generated LOTOS code.
- The pragma "`!pointer`" specifies that type $T$ must be implemented by a pointer in C.
- The pragma "`!nopointer`" specifies that type $T$ must not be implemented by a pointer in C, i.e., it must have an unboxed implementation.
- The pragma "`!bits` *nat*" specifies a maximal value $2^{nat}$ for the number of elements of type $T$, meaning that each of these elements will be implemented in C using at most *nat* bits.

  Note: This pragma is implemented by inserting a macro "`CAESAR_ADT_HASH_`$T'$ `-`*nat*" in the generated C code, where $T'$ is the name of the C type implementing type $T$. For details, see entries #623 and #1250 of file "`$CADP/HISTORY`".
- The pragma "`!card` *nat*" specifies a maximal value *nat* for the number of elements of type $T$.

  Note: This pragma is implemented by inserting a macro "`CAESAR_ADT_HASH_`$T'$ *nat*" in the generated C code, where $T'$ is the name of the C type implementing type $T$. For details, see entries #623 and #1250 of file "`$CADP/HISTORY`".

In pragmas "`!implementedby "C:..."`", "`!implementedby "LOTOS:..."`", "`!comparedby "C:..."`", "`!iteratedby "C:...", "C:..."`", and "`!printedby "C:..."`", the prefixes "`C:`" and "`LOTOS:`" are case-sensitive. Other forms, such as "`c:`" and "`Lotos:`" are rejected.

## 5.7 Predefined function declarations

For the basic data types (Boolean, natural number, integer, real number, character, string), a number of predefined functions are automatically available. See Annex C for the list of these predefined functions.

For the non-basic data types, predefined functions are generated according to the specified "**with**" clauses. We split non-basic data types into various sub-categories:

- Singleton types, consisting of a single constructor, either without parameters or whose parameters are all of singleton types.

- Enumerated types, consisting of several constructors, either without parameters or whose parameters are all of singleton types:

  **type** $T$ **is**
  $C_0, ..., C_n$
  **end type**

- Cascade types, consisting of several constructors, at least one of which has parameters, but only of singleton and/or enumerated types.
- Numeral types, consisting of several constructors, one of which has a parameter of this numeral type; The constructors may have additional parameters, provided they are all of singleton types.
- Set types $T$ declared as:

  **type** $T$ **is**
  **set of** $T'$
  **end type**

- List and sorted list types $T$ declared as:

  **type** $T$ **is**
  [**sorted**] **list of** $T'$
  **end type**

- Array types $T$ declared as:

  **type** $T$ **is**
  **array** [$m$..$n$] **of** $T'$
  **end type**

- Range types $T$ declared as:

  **type** $T$ **is**
  **range** $m$..$n$ **of** $T'$
  **end type**

- Predicate types $T$:

  **type** $T$ **is**
  $X : T'$ [**where** *value expression*]
  **end type**

  The type expression "**type** $T$ **is** $X$ : $T'$ **end type**" is equivalent to "**type** $T$ **is** $X$ : $T'$ **where true end type**". The only difference is that the latter definition issues a warning "*condition always true*", whereas the former does not.

- All other non-basic types $T$, including record-like types, union-like types, etc.

The following table shows the LNT predefined constructors and functions that can be created for non-basic data types. The functions marked by a star are generated automatically. The other functions are optional and must be generated by specifying the relevant "**with**" clause in the data type declaration.

| Function | Profile | Supported data types $T$ |
|---|---|---|
| `==`, `=` | $T, T \to$ Bool | all types |
| `<>`, `!=` | $T, T \to$ Bool | all types |
| `<`, `<=`, `>`, `>=` | $T, T \to$ Bool | all types |
| `ord` | $T \to$ Nat | all types |
| `val` | Nat $\to T$ | singleton, enumerated, range |
| `first` | $\to T$ | singleton, enumerated, cascade, range |
| `last` | $\to T$ | singleton, enumerated, cascade, range |
| `succ` | $T \to T$ | enumerated, cascade, numeral, range |
| `pred` | $T \to T$ | enumerated, cascade, numeral, range |
| `get` functions | $T \to U$ | all types but singleton, enumerated, range |
| `set` functions | $U, T \to T$ | all types but singleton, enumerated, range |
| `nil`$^*$ | $\to T$ | set, list, sorted list |
| `cons`$^*$ | $T', T \to T$ | set, list, sorted list |
| `insert`$^*$ | $T', T \to T$ | set, list, sorted list |
| `empty` | $T \to$ Bool | set, list, sorted list |
| `length` | $T \to$ Nat | set, list, sorted list |
| `member` | $T', T \to$ Bool | set, list, sorted list |
| `element` | $T$, Nat $\to T'$ | set, list, sorted list |
| `delete` | $T', T \to T$ | set, list, sorted list |
| `remove` | $T', T \to T$ | set, list, sorted list |
| `head` | $T \to T'$ | set, list, sorted list |
| `tail` | $T \to T$ | set, list, sorted list |
| `union` | $T, T \to T$ | set, list, sorted list |
| `inter` | $T, T \to T$ | set, sorted list |
| `minus` | $T, T \to T$ | set, sorted list |
| `diff` | $T, T \to T$ | set, sorted list |
| `reverse` | $T \to T$ | (unsorted) list |
| `append` | $T', T \to T$ | (unsorted) list |
| `subset` | $T, T \to$ Bool | set |
| $T$ *(array constructor)*$^*$ | $T' \to T$ | array |
| $T$ *(array constructor)*$^*$ | $T'$, ..., $T' \to T$ | array |
| $T$ *(conversion to subtype)*$^*$ | $T' \to T$ | range, predicate (partial function) |
| $T$ *(identity)*$^*$ | $T \to T$ | range, predicate |
| $T'$ *(conversion to parent type)*$^*$ | $T \to T'$ | range, predicate |

These predefined functions over non-basic types are defined as follows:

- Comparison operators can be generated for all types $T$. All these operators have the same profile: `T , T -> BOOL`.

  Equality relations correspond to structural equivalence between values of type $T$.

  Order relations correspond to the underlying lexicographic order (which is a total order) over values of type $T$ considered as algebraic terms (constructors are ordered by their occurrence of declaration in the LNT type definition — in the case of sets, lists, and sorted lists, the *nil* constructor is considered to be smaller than the *cons* constructor). Note therefore that, in the case of lists or sets, these comparison operators do not correspond to (list or set) inclusions (which are partial orders).

The inequality operator "`!=`" is translated into a LOTOS operator "`/=`" since the character "`!`" cannot be used in LOTOS special identifiers.

- The function "`ord :` $T$ `-> NAT`" can be generated for all types $T$.

  If $T$ is not a range type, `ord (X)` returns, for each element $X$ of type $T$, the order number of the constructor of $X$, the first constructor being numbered 0 and the last constructor being numbered $n-1$ where $n$ is the number of constructors of $T$.

  If $T$ is a range type of the form "**range** $m$`..`$n$", `ord (X)` returns the order number of $X$ in that range, the lower bound $m$ being numbered 0 and the upper bound $n$ being numbered $n-m$.

- The function "`val : NAT ->` $T$" can be generated only when "**ord**" is injective, i.e., only when $T$ is an enumerated type or a range type.

  For each value $X$ of type $T$, `val (ord (X)) = X`.

- The functions "`first : ->` $T$" and "`last : ->` $T$" can be generated only when $T$ is an enumerated type, a cascade type, or a range type.

  These functions return, respectively, the smallest and greatest values of type $T$. For enumerated and range types, `first` = `val`(`ord`(0)) and `last` = `val`(`ord`($n-1$)), where $n$ is the number of constructors of $T$.

- The functions "`succ :` $T$ `->` $T$" and "`pred :` $T$ `->` $T$" can be generated only when $T$ is an enumerated type, a cascade type, or a range type.

  These functions return, respectively, the successor and the predecessor of a value of type $T$. Note that the greatest (resp. smallest) element of $T$ is its own successor (resp. predecessor).

- For all types $T$ except enumerated and range types, when "`get`" appears in the list of requested functions given in the "**with**" clause of type $T$, one or several LOTOS functions (named "`get`" functions) will be generated, which will enable the use of *field selection* notations for values of type $T$ (see the syntax of expressions in Section 7.2).

  For each constructor $C$ of $T$, for each argument $f$ (of type $U$) of constructor $C$, a (partially defined) LOTOS function named "`GET_`$f$ `:` $T$ `->` $U$" will be generated. For each value $X$ of type $T$, if $X$ has the form $C(...)$, where $C$ is a constructor with an argument named $f$, then `GET_`$f$ $(X)$ returns the value of $f$, otherwise it is undefined.

- For all types $T$ except enumerated and range types, when "`set`" appears in the list of requested functions given in the "**with**" clause of type $T$, one or several LOTOS functions (named "`set`" functions) will be generated, which will enable the use of *field update* notations for values of type $T$ (see the syntax of expressions in Section 7.2).

  For each constructor $C$ of $T$, for each argument $f$ (of type $U$) of constructor $C$, a (partially defined) LOTOS function named "`SET_`$f$ `:` $U$`,` $T$ `->` $U$" will be generated. For each value $X$ of type $T$ and each value $Y$ of type $U$, if $X$ has the form $C(...)$, where $C$ is a constructor with an argument named $f$, then `SET_`$f$ $(Y, X)$ returns the value of $X$ in which argument $f$ has been replace by $Y$, otherwise it is undefined.

- The **list**, **sorted list**, and **set** types are very similar: the three of them have two constructors, "`nil`" and "`cons`" and an operation "`insert`". In **list**, `insert` is a synonym of `cons`, but is not a constructor. In **sorted list**, `insert` enables one to add an element to a list, still preserving the invariant that list elements are sorted. In **set**, `insert` enables one to add an element to a set, still preserving the invariant that set elements are sorted and each element of a set has at most one occurrence. `insert` can be used in expressions, but cannot be used in patterns since it is not a type constructor.

Those four types also differ in the sets of predefined functions that can be generated using the "**with**" clause (see items below).

- The function "`empty :` $T \to$ `BOOL`" can be generated for all types $T$ of the form "**set of** $T'$", "**list of** $T'$", or "**sorted list of** $T'$".

  For each value $X$ of type $T$, `empty (X)` returns `true` if $X$ is empty.

- The function "`card :` $T \to$ `NAT`" can be generated for all types $T$ of the form "**set of** $T'$".

  For each value $X$ of type $T$, `card (X)` returns the number of elements in $X$.

- The function "`length :` $T \to$ `NAT`" can be generated for all types $T$ of the form "**list of** $T'$", or "**sorted list of** $T'$".

  For each value $X$ of type $T$, `length (X)` returns the number of elements in $X$.

- The function "`member :` $T'$`,` $T \to$ `BOOL`" can be generated for all types $T$ of the form "**set of** $T'$", "**list of** $T'$", or "**sorted list of** $T'$".

  For each value $X$ of type $T'$ and $Y$ of type $T$, `member (X, Y)` returns `true` if $X$ occurs in $Y$.

- The function "`element :` $T$`,` `NAT` $\to T'$" can be generated for all types $T$ of the form "**set of** $T'$", "**list of** $T'$", or "**sorted list of** $T'$".

  For each value $X$ of type $T$ and $N$ of type `NAT`, `element (X, N)` returns the $N$-th element of $X$. An error occurs if $N$ is zero or greater than `length (X)` (for list and sorted list types) or `card (X)` (for set types).

- The function "`delete :` $T'$`,` $T \to T$" can be generated for all types $T$ of the form "**set of** $T'$", "**list of** $T'$", or "**sorted list of** $T'$".

  For each value $X$ of type $T'$ and $Y$ of type $T$, `delete (X, Y)` returns a copy of $Y$ from which the first occurrence of $X$ (if any) has been suppressed. If $Y$ does not contain any occurrence of $X$, then `delete (X, Y)` returns $Y$ unchanged.

- The function "`remove :` $T'$`,` $T \to T$" can be generated for all types $T$ of the form "**set of** $T'$", "**list of** $T'$", or "**sorted list of** $T'$".

  For each value $X$ of type $T'$ and $Y$ of type $T$, `remove (X, Y)` returns a copy of $Y$ from which all occurrences of $X$ (if any) have been suppressed. If $Y$ does not contain any occurrence of $X$, then `remove (X, Y)` returns $Y$ unchanged. Note that if $T$ is a set type, the functions `delete` and `remove` coincide since each element of type $T'$ has at most one occurrence in $Y$.

- The function "`head :` $T \to T'$" can be generated for all types $T$ of the form "**set of** $T'$", "**list of** $T'$", or "**sorted list of** $T'$".

  For each value $X$ of type $T$, `head (X)` returns the first element of $X$. An error occurs if $X =$ `nil`.

- The function "`tail :` $T \to T$" can be generated for all types $T$ of the form "**set of** $T'$", "**list of** $T'$", or "**sorted list of** $T'$".

  For each value $X$ of type $T$, `tail (X)` returns a copy of $X$ from which the first element has been removed. An error occurs if $X =$ `nil`.

- The function "`union :` $T$ `,` $T \rightarrow T$" can be generated for all types $T$ of the form "**set of** $T'$", "**list of** $T'$", or "**sorted list of** $T'$".

  For each values $X$ and $Y$ of type $T$, `union (X, Y)` (or "`X union Y`" in infix notation) returns: if $T$ is a set type, the set union $X \cup Y$; if $T$ is an unsorted list type, the concatenation of lists $X$ and $Y$; if $T$ is a sorted list type, the sorted merge of $X$ and $Y$.

- The function "`inter :` $T$ `,` $T \rightarrow T$" can be generated for all types $T$ of the form "**set of** $T'$" or "**sorted list of** $T'$".

  For each values $X$ and $Y$ of type $T$, `inter (X, Y)` (or "`X inter Y`" in infix notation) returns: if $T$ is a set type, the set intersection $X \cap Y$; if $T$ is a sorted list type, the multiset-like sorted intersection of $X$ and $Y$ (namely, if $X$ and $Y$ contain respectively $n$ and $m$ occurrences of some element $z$, then `inter (X, Y)` contains exactly $\min(n, m)$ occurrences of $z$).

- The function "`minus :` $T$ `,` $T \rightarrow T$" can be generated for all types $T$ of the form "**set of** $T'$" or "**sorted list of** $T'$".

  For each values $X$ and $Y$ of type $T$, `minus (X, Y)` (or "`X minus Y`" in infix notation) returns: if $T$ is a set type, the asymmetric set difference, that is the set of elements present in $X$ and not present in $Y$; if $T$ is a sorted list type, the asymmetric multiset-like sorted difference between $X$ and $Y$ (namely, if $X$ and $Y$ contain respectively $n$ and $m$ occurrences of some element $z$, then `minus (X, Y)` contains exactly $\max(0, n - m)$ occurrences of $z$).

- The function "`diff :` $T$ `,` $T \rightarrow T$" can be generated for all types $T$ of the form "**set of** $T'$" or "**sorted list of** $T'$".

  For each values $X$ and $Y$ of type $T$, `diff (X, Y)` (or "`X diff Y`" in infix notation) returns: if $T$ is a set type, the symmetric set difference, that is the set of elements present in one of $X$ or $Y$ and not present in the other; if $T$ is a sorted list type, the symmetric multiset-like sorted difference between $X$ and $Y$ (namely, if $X$ and $Y$ contain respectively $n$ and $m$ occurrences of some element $z$, then `diff (X, Y)` contains exactly $\max(m - n, n - m)$ occurrences of $z$). In both cases, `diff (X, Y)` is equal to `minus (X, Y) union minus (Y, X)`.

- The function "`reverse :` $T \rightarrow T$" can be generated for all types $T$ of the form "**list of** $T'$".

  For each value $X$ of type $T$, `reverse (X)` returns a copy of $X$ in which the elements occur in reverse order.

- The function "`append :` $T'$ `,` $T \rightarrow T$" can be generated for all types $T$ of the form "**list of** $T'$".

  For each value $X$ of type $T'$ and $Y$ of type $T$, `append (X, Y)` returns a copy of $Y$ in which element $X$ has been added in the last position.

- The function "`subset :` $T$ `,` $T \rightarrow$ `BOOL`" can be generated for all types $T$ of the form "**set of** $T'$".

  For each values $X$ and $Y$ of type $T$, `subset (X, Y)` (or "`X subset Y`" in infix notation) returns `true` if all members of $X$ are members of $Y$.

- The function "$T$ `:` $T' \rightarrow T$" is generated for all types $T$ of the form "**array** `[` m `..` n `]` **of** $T'$". It enables to construct an array whose items are all set to the same value of type $T'$ passed as argument.

- The function "$T$ `:` $T'$ `, ... ,` $T' \rightarrow T$" is generated for all types $T$ of the form "**array** `[` m `..` n `]` **of** $T'$". It enables to construct an array whose $i$th item is defined by the $i$th of the $n - m + 1$ arguments of type $T'$.

- Note: The definition of a range or predicate type $T$ will generate LOTOS functions that implement conversion to subtype, identity, and conversion to parent type. Indentity functions are only generated for type checking reasons (namely, to easily produce a LOTOS program that will type check correctly) and are not intended to be directly invoked from LNT.
- Note: The definition of an array type $T$ will generate LOTOS functions that implement access and modification of array elements: the accessor function "`ARRAY_GET :` $T$`, NAT ->` $T'$" and the modifier function "`ARRAY_SET :` $T$`, NAT,` $T'$ `->` $T$", where $T'$ is the type of the array elements. These functions should not be invoked directly from LNT.

We summarize here the constraints that apply to predefined functions over non-basic types:

(PF1) *This constraint was removed in January 2023; see item* `#XXXX` *in the* `$CADP/HISTORY` *file.*

(PF2) The function "`val`" can be generated only for enumerated types and range types.

(PF3) The functions "`first`" and "`last`" can be generated for enumerated types and range types only.

(PF4) The functions "`get`" and "`set`" can be generated for all but enumerated types and range types.

(PF5) The functions "`empty`", "`member`", "`element`", "`delete`", "`remove`", "`head`", "`tail`", and "`union`" can be generated for set types, list types, and sorted list types. Function "`union`" can be used in both prefix and infix notation.

(PF6) The functions "`inter`", "`minus`", and "`diff`" can be generated for set types and sorted list types. The three of these functions can be used in both prefix and infix notation.

(PF7) The functions "`reverse`" and "`append`" can be generated for (unsorted) list types only.

(PF8) The functions "`card`" and "`subset`" can be generated for set types only. Function "`subset`" can be used in both prefix and infix notation.

(PF9) The function "`length`" can be generated for list types and sorted list types only.

Users are allowed to define operations with same names and types as the above, preventing the use of the corresponding "**with**" clauses. Beware however that some operations present in "**with**" clause may depend on others, which are then generated implicitly by LNT2LOTOS. For instance, if a set type $T$ has a "**with** `subset`" clause, then an operation "`member`" will also be generated for $T$, thus preventing users from defining their own version of "`member`".

Some operations should ensure properties, which cannot be checked automatically but are key for program correctness. In particular:

- Obviously, all operations named `==` or `=` of profile "$T, T \rightarrow$ Bool" should be equivalence relations (i.e., they should be transitive, reflexive, and symmetric). All operations named `!=` or `<>` of profile $T, T \rightarrow$ Bool" should implement the negation of `==` or `=`.
- Operations named `<` of profile "$T, T \rightarrow$ Bool" should be total orders (i.e., for every every two values $v_1$ and $v_2$ of type $T$, $v_1$ `<` $v_2$ iff not $v_2$ `<` $v_1$) if they are used for **set** or **sorted list** types. Otherwise, for instance, if "`<:` $T$, $T \rightarrow$ Bool" is not a total order and a type $T'$ is defined as "**set of** $T$ **with** `==`", then the operation `==` defined by LNT2LOTOS (which returns true only if its operands are structurally identical) will not be an equivalence relation, because the representation of the elements of $T'$ is not canonical (i.e., identical elements may have structurally different representations).

If all user-defined operations named `==`, `=`, `!=`, `<>`, `<`, `<=`, `>`, and `>=` of profile "$T, T \rightarrow$ Bool" satisfy the above properties, then those generated by LNT2LOTOS also ensure the above properties.

## 5.8 Predefined function pragmas

The LNT syntax enables function pragmas to be attached to predefined functions in the "**with**" clauses:

(PFP1) In the list *predefined_function_pragma*$_1$...*predefined_function_pragma*$_n$ of each predefined function declaration, there should be at most one pragma of each kind (i.e., there cannot be two "`!external`" pragmas, nor two "`!implementedby "..."`"/"`!implementedby "C:..."`" pragmas, nor two "`!implementedby "LOTOS:..."`" pragmas, etc.)

(PFP2) *This constraint was removed in September 2022; see item* `#2822` *in the* `$CADP/HISTORY` *file.*

(PFP3) *This constraint was removed in September 2022; see item* `#2822` *in the* `$CADP/HISTORY` *file.*

The pragmas "`!implementedby "..."`", "`!implementedby "C:..."`" and "**implementedby** `"LOTOS:..."`" for the predefined functions "`get`" and "`set`" enable one to control the names of the C functions and LOTOS operations following the same rules as for standard functions (see Section 7.6).

Predefined functions support a pragma `!external`. In that case, the code of the corresponding function is not generated automatically, and its definition must be provided as external code, as explained in chapter 7 for general functions.

In pragmas "`!implementedby "C:..."`" and "`!implementedby "LOTOS:..."`", the prefixes `C` and `LOTOS` are case-insensitive and can alternatively be written using any combination of upper- and lower-case characters, such as `c`, `lotos`, `Lotos`, etc.

## 5.9 Module "with" clauses

A "**with**" clause in a module $M$ provides a list of predefined functions to be declared automatically in each type definition of $M$ (see Section 4.2); this list, given at the module level, is subsequently enriched by the list declared in the "**with**" clause of each type definition in $M$.

Notice that the declarations of predefined functions in the "**with**" clause of a module do not allow pragmas, since this could create problems. For instance, a pragma "`!implementedby "C:..."`" for a predefined function declared by the "**with**" clause of a module $M$ leads to name conflicts for the generated C functions if $M$ contains several type definitions.

# Chapter 6

# Channel definitions in LNT

## 6.1 Notations

This chapter uses the BNF notations defined in Section 3.1 and the non-terminals defined in Chapter 5. The following additional conventions are used:

- $\Gamma$ is a channel identifier
- $T$ is a type identifier
- $X$ is a variable identifier

## 6.2 Syntax

| | | | |
|---|---|---|---|
| *channel_definition* | ::= | **channel** $\Gamma$ **is** [**raise**]<br>*channel_profile*$_0$ ,<br>... ,<br>*channel_profile*$_n$<br>**end channel** | *channel definition* |
| *channel_profile* | ::= | ( *profile_parameters*$_1$ , ... , *profile_parameters*$_n$ ) | *channel profile* |
| *profile_parameters* | ::= | $X_0$ , ... , $X_n$ : $T$ | *profile parameter list* |

## 6.3 Channels

A channel defines a set of channel profiles. If a channel has more than one profile, it is called *overloaded*.

(CH1) The channel names must be pairwise distinct.

(CH2) The profiles in a channel definition must be pairwise distinct.

(CH3) There exists a predefined channel identifier (noted "**none**") that is implicitly declared at the top level and is visible in each LNT module. This channel is defined as follows:

**channel none is** () **end channel**

This channel can be used to declare events that are used to perform pure synchronization (without offers). This channel must not be redeclared explicitly.

(CH4) There exists a predefined channel identifier (noted "**exit**") that is implicitly declared at the top level and is visible in each LNT module. This channel is defined as follows:

**channel exit is raise** () **end channel**

This channel can be used to declare events that are used as exceptions without parameters. This channel must not be redeclared explicitly.

(CH5) We call "**raise**" channel any channel whose definition contains the "**raise**" keyword. An event can be used as an exception iff it is declared with a "**raise**" channel, and it can be used in a communication iff it is not declared with a "**raise**" channel. However it is not yet possible for users to declare their own "**raise**" channels, so that "**exit**" is the only "**raise**" channel available so far. An event declared with "**any**" cannot be used as an exception. [checked by LNT2LOTOS]

(CH6) To avoid confusion with the keyword "**any**" (always written in lower case), user-defined channel names should be distinct from "**ANY**", "**Any**", or any identifier that is identical to "**any**" modulo case-insensitive string comparison. [checked by LNT2LOTOS]

## 6.4 Channel profiles

A channel profile is a possibly empty list of named parameters.

(CP1) The types $T$ occurring in channel profiles must have been declared, unless they are predefined types.

(CP2) The variable identifiers in a channel profile must be pairwise distinct.

(CP3) In the same channel definition, profile parameters declared with the same variable identifier should have the same type.

## 6.5 Gate and exception events

LNT has a concept of "*event*", which serves for two purposes:

- Events can be used to model gates. As in LOTOS, gates can be used for input/output communication or synchronization. This can only occur in LNT processes, since LNT constructors, functions, and procedures perform only local calculations and are not allowed to engage in communication or synchronization.
- Events can also be used to model exceptions. This can occur in LNT functions, procedures, and processes, all of which can trigger exceptions using the "**raise**" construct. Constructors are not allowed to raise exceptions, as these operations are assumed to be total. Thus, all the events present in LNT functions and procedures represent exceptions.

LNT supports the concept of exceptions in the following way:

- In the current version of LNT2LOTOS, exceptions cannot carry value parameters. Thus, every exception must be declared with channel "**exit**". This constraint may be relaxed in the future.
- In the current version of LNT2LOTOS, exceptions are uncatchable: when an exception is raised at runtime, the executed program prints a message and stops. Thus, so far, LNT exceptions can only be used to model unwanted conditions that provoke a fatal termination of the entire system.
- LNT follows the *checked exception* paradigm, meaning that the exceptions raised in a routine are not global objects, but must be declared as formal parameters of this routine. Syntactically, such parameters are declared between square brackets.
- When a function, a procedure, or a process that raises exceptions is called, its formal exception parameters must be instantiated with actual exceptions, in the same way as passing arguments to a function call, but still using square brackets. This is done by inserting a bracketed list of actual exceptions right after the routine identifier, e.g., "`next_element [end_of_list] (x)`", "`sum [overflow, underflow] (x, y)`", etc., and "`x sum [overflow, underflow] y`" in the particular case of an infix function.
- The two latter items merely and straightforwardly extend the rules that exist for gates in LOTOS and LNT processes: each process must be declared with formal event parameters, which have to be instantiated with actual events when the process is called.
- In the current version of LNT2LOTOS, it is not allowed to freely mix both types of events: an exception cannot be used where a gate is expected, and vice-versa.
- However, LNT2LOTOS does not statically detect the case where, in a process call, a formal exception parameter is instantiated with an actual gate. In such case, LNT2LOTOS will emit no warning and generate LOTOS code that compiles properly. Unfortunately, at run-time, the parameter substitution will not take place. Such an issue does not occur in function and procedure calls.

## 6.6 Predefined events

In addition to user-defined events, there are three special events in LNT:

- The *internal* (or *invisible*) event is noted "**i**" in LNT and LOTOS. This event corresponds to the notion of invisible action noted $\tau$ in concurrency-theory textbooks. It is a gate (i.e., not an exception), but it cannot be actually used for communication or synchronization. The channel of this event is "**none**".
- The *continuation* (or *successful termination*) event is noted "$\delta$". This event does not appear explicitly in the syntax of LOTOS and LNT, but appears in the dynamic semantics of LOTOS and LNT processes. This event appears each time a behaviour terminates, yielding the control to another behaviour to be executed in sequence. For example, the "**null**" behaviour of LNT generates an action on the event "$\delta$". The channel of this event is "**any**".
- The *anonymous* event is noted "**unexpected**" in the concrete LNT syntax (notice that this is not a reserved keyword, but a predefined event identifier) and $\xi$ in the semantics of LNT. This event is an exception declared implicitly at the top level and thus should never occur in event declarations. The channel of this event is "**exit**".

## 6.7 Compatible events

In LNT routines, formal event parameters can be typed by a channel (following the ideas of [Gar95]) or declared as untyped (like LOTOS gates) using the "**any**" keyword.

We therefore define a compatibility relation between events, so as to determine when a formal event parameter $E_1$ can be instantiated by an actual event $E_2$.

Two events $E_1$ and $E_2$ are *compatible* if and only if:

- $E_1$ and $E_2$ are both declared as exceptions or are both declared as gates, and
- $E_1$ and $E_2$ are both untyped (i.e., declared with "**any**") or are both declared with the same channel $\Gamma$.

The former rule expresses that a formal event declared as an exception (resp. as a gate) must be instantiated by an actual event declared as an exception (resp. as a gate). Consequently, the actual event parameters used in a function call must be exceptions.

The latter rule is based upon "name equivalence" for channels, which simplifies the static semantics and fits smoothly into the philosophy of LOTOS; the motivation for this choice is given more explicitly in [Gar95].

# Chapter 7

# Function definitions in LNT

## 7.1 Notations

This chapter uses the BNF notations defined in Section 3.1 and the non-terminals defined in Chapters 5 and 6.

The following additional conventions are used:

- $F$ is a function identifier
- $X$ is a variable identifier
- $I$ is a statement
- $V$ is an expression
- $P$ is a pattern
- $L$ is a loop label
- $E$ is an event identifier (which may denote either a gate or an exception)

The present chapter gives syntactic and semantic definitions for functions and procedures. Many of these definitions are reused later for processes in Chapter 8, since processes are a superset of procedures. For conciseness, the definitions of the present chapter are generalized to the case of processes whenever appropriate.

## 7.2 Syntax

*function_definition* ::= **function** $F$ [ `[`$\mathit{formal\_events}_0$ `,` ... `,` $\mathit{formal\_events}_m$`]` ]
[ `(`$\mathit{formal\_parameters}_1$ `,` ... `,` $\mathit{formal\_parameters}_n$`)` ] [ `:`$T$ ] **is**
$\mathit{function\_pragma}_1 ... \mathit{function\_pragma}_l$
$\mathit{precondition}_1 ... \mathit{precondition}_j$
$\mathit{postcondition}_1 ... \mathit{postcondition}_k$

| | | | |
|---|---|---|---|
| | | [ $I_0$ ] | |
| | | **end function** | *function definition* |
| *formal_events* | ::= | *event_declaration* | *formal events* |
| *event_declaration* | ::= | $E_0, \ldots, E_n : \Gamma$ | *typed event declaration* |
| | \| | $E_0, \ldots, E_n$ : **any** | *untyped event declaration* |
| *formal_parameters* | ::= | *parameter_mode* $X_0, \ldots, X_n : T$ | *formal parameters* |
| *parameter_mode* | ::= | [ **in** ] | *input formal parameter* |
| | \| | **in var** | *input formal parameter used as local variable* |
| | \| | **out** | *output formal parameter* |
| | \| | **out var** | *output formal parameter used as local variable* |
| | \| | **in out** | *input / output formal parameter* |
| *precondition* | ::= | **require** $V$ [ **raise** $E$ [ `()` ] ]; | *precondition* |
| *postcondition* | ::= | **ensure** $V$ [ **raise** $E$ [ `()` ] ]; | *postcondition* |
| *function_pragma* | ::= | `!external` | *external function* |
| | \| | `!implementedby "`[(`C` \| `LOTOS`)`:`]*name*`"` | *C or* LOTOS *name scheme* |
| $I$ | ::= | **null** | *no effect* |
| | \| | $I_1$ `;` $I_2$ | *sequential composition* |
| | \| | **return** [ $V$ ] | *return* |
| | \| | **raise** $E$ [ `()` ] | *exception raise* |
| | \| | **assert** $V$ [ **raise** $E$ [ `()` ] ] | *assertion* |
| | \| | $X$ `:=` $V$ | *assignment* |
| | \| | $X$ `[`$V_0$`]` `:=` $V_1$ | *array element assignment* |

| | |
|---|---|
| $\mid$ [ **eval** ] [ $X$ `:=` ] $F$ [ `[`*actual_events*`]` ] `(`*actual_parameter*$_1$ `,`...`,` *actual_parameter*$_n$`)` | *procedure call* |
| $\mid$ **var** *var_declaration*$_0$ `,`...`,` *var_declaration*$_n$ **in** <br> $I_0$ <br> **end var** | *variable declaration* |
| $\mid$ **case** $V_0$ `,`...`,` $V_\ell$ <br> [ **var** *var_declaration*$_0$ `,`...`,` *var_declaration*$_n$] **in** <br> *match_clause*$_0$ `->` $I_0$ <br> `|` ... <br> `|` *match_clause*$_m$ `->` $I_m$ <br> **end case** | *case statement* |
| $\mid$ **if** $V_0$ **then** $I_0$ <br> [ **elsif** $V_1$ **then** $I_1$ <br> ... <br> **elsif** $V_n$ **then** $I_n$ ] <br> [ **else** $I_{n+1}$ ] <br> **end if** | *conditional statement* |
| $\mid$ **loop** <br> $I_0$ <br> **end loop** | *forever loop* |
| $\mid$ **loop** $L$ **in** <br> $I_0$ <br> **end loop** | *breakable loop* |
| $\mid$ **while** $V$ **loop** <br> $I_0$ <br> **end loop** | *while loop* |
| $\mid$ **while** $V$ **loop** $L$ **in** <br> $I_0$ <br> **end loop** | *breakable while loop* |
| $\mid$ **for** $I_0$ **while** $V$ **by** $I_1$ **loop** <br> $I_2$ <br> **end loop** | *for loop* |
| $\mid$ **for** $I_0$ **while** $V$ **by** $I_1$ **loop** $L$ **in** <br> $I_2$ <br> **end loop** | *breakable for loop* |
| $\mid$ **break** $L$ | *loop break* |
| $\mid$ **use** $X_0$ `,`...`,` $X_n$ | *variable use* |
| $\mid$ **access** $E_0$ `,`...`,` $E_n$ | *event access* |

| | | | |
|---|---|---|---|
| *var_declaration* | ::= | $X_0$ , ... , $X_n$ : $T$ | *variable list* |
| | | | |
| *actual_events* | ::= | $E_1$ , ... , $E_n$ | *positional style* |
| | \| | $E_{formal,1}$ -> $E_{actual,1}$ , ... , $E_{formal,n}$ -> $E_{actual,n}$ [ , **...** ] | *named style* |
| | | | |
| *actual_parameter* | ::= | $V$ | *actual parameter "in"* |
| | \| | ?$X$ | *actual parameter "out"* |
| | \| | !?$X$ | *actual parameter "in out"* |
| | | | |
| *match_clause* | ::= | $P_0$ , ... , $P_\ell$ [ **where** $V_0$ ] \| ... \| $P_n$ [ **where** $V_n$ ] | *match clause* |
| | \| | **any** , ... , **any** [ **where** $V$ ] | *wildcard* |
| | | | |
| $P$ | ::= | $X$ | *variable* |
| | \| | **any** $T$ | *wildcard* |
| | \| | $X$ **as** $P_0$ | *aliasing* |
| | \| | $C$ [ ( $P_0$ , ... , $P_n$ ) ] | *constructed pattern* |
| | \| | $P_1$ $C$ $P_2$ | *infix constructed pattern* |
| | \| | $F$ [ ( $P_0$ , ... , $P_n$ ) ] | *constant pattern* |
| | \| | $P_1$ $F$ $P_2$ | *infix constant pattern* |
| | \| | $P_0$ **of** $T$ | *type coercion* |
| | \| | ( $P$ ) | *parenthesized pattern* |
| | \| | { $P_1$ , ... , $P_n$ } | *list pattern* |
| | | | |
| $V$ | ::= | $X$ | *variable* |
| | \| | $X$ . **in** | *input parameter value (in postcondition only)* |
| | \| | $X$ . **out** | *output parameter value (in postcondition only)* |
| | \| | **result** | *function result (in postcondition only)* |
| | \| | $C$ [ ( $V_1$ , ... , $V_n$ ) ] | *constructor call* |
| | \| | $V_1$ $C$ $V_2$ | *infix constructor call* |
| | \| | $F$ [ [ *actual_events* ] ] [ ( $V_1$ , ... , $V_n$ ) ] | *function call* |
| | \| | $V_1$ $F$ [ [ *actual_events* ] ] $V_2$ | *infix function call* |
| | \| | $V$ . [ [ $E$ ] ] *field* | *field selection* |

| | | |
|---|---|---|
| \| | $V$ `.[` [ `[`$E$`]` ] `{`$field_0$`->`$V_0$`,...,`$field_n$`->`$V_n$`}` | *field update* |
| \| | $V_0$ `[` $V_1$ `]` | *array element access* |
| \| | $V$ **of** $T$ | *type coercion* |
| \| | `(`$V$`)` | *parenthesized expression* |
| \| | `{`$V_1$`,...,`$V_n$`}` | *list expression* |

## 7.3 Resolution of syntactic ambiguities

### Ambiguity 1

In a statement $I$ having the form "$X$ `:=` $Z$ [ `[`*actual_events*`]` ] `(`$V_1, ..., V_n$`)`", where $Z$ is an identifier and where $V_1, ..., V_n$ are value expressions, there is a syntactic ambiguity, as statement $I$ can be parsed either using the assignment rule (in such case "$Z$ [ `[`*actual_events*`]` ] `(`$V_1, ..., V_n$`)`" is parsed as an function-call expression) or using the procedure-call rule.

This ambiguity is solved on the semantic level. Indeed, identifier $Z$ must be a function identifier and cannot be a procedure identifier, because $Z$ is invoked here with "**in**" or "**in var**" parameters only, whereas a procedure has at least one "**out**", "**out var**, or "**in out**" parameter. Thus, statement $I$ must be interpreted as an assignment to $X$ of a call to function $Z$.

### Ambiguity 2

In a pattern $P$ having the form "$Z$ [ `(`$P_0$`,...,`$P_n$`)` ]", where $Z$ is an identifier, there is a syntactic ambiguity between the constructed pattern rule and the constant pattern rule.

Similarly, in a pattern $P$ having the form "$V_1$ $Z$ $V_2$", there is also a syntactic ambiguity between the constructed pattern rule and the constant pattern rule.

This ambiguity is resolved on the semantic level. The pattern $P$ is considered to be a constructed pattern if at least one of the patterns $P_0, ..., P_n$ is not a constant pattern or if there exists a constructor $Z$ whose arguments have the same types as $P_0, ..., P_n$ and whose result has the same type as $P$.

### Ambiguity 3

In a value expression $V$ having the form "$Z$", where $Z$ is an identifier, there is a syntactic ambiguity between a variable, a call to a constructor without parameter, and a call to a function without parameter.

This ambiguity is resolved on the semantic level. If a variable named $Z$ is declared in the current context, then $V$ is considered a variable. If not, $V$ is assumed to be a constructor call or a function call.

Thus, priority is given to variable identifiers with respect to constructor and function identifiers.

Notice that a variable $Z$ can coexist with functions and/or constructors having the same name and without parameter. In such cases, the expression $Z$ is understood as referencing the variable, but it is always possible to call the functions and/or constructors by having their names followed by empty parentheses, i.e. $Z$`()`.

In expressions, LNT2LOTOS makes no distinction between a function call $F$ and a constructor call $C$. This ambiguity is solved by CÆSAR/CÆSAR.ADT.

## Precedence rules

The following precedence rules apply to patterns and value expressions. The precedence of operators (from highest to lowest) is:

- in patterns: prefix constructed patterns, **of** (type coercion), infix constructed patterns (see next paragraph), **as** (aliasing)
- in value expressions: prefix function calls, array element accesses, dotted notations (field selection and field update), **of** (type coercion), infix function calls (see next paragraph), and constructor calls

The precedence of operators (from highest to lowest) in infix constructed patterns and infix function calls is:

- "**of**", "`.`" field selection and update
- infix operators not listed below
- "`**`"
- "`*`", "`/`", "`div`", "`mod`", "`rem`"
- "`+`", "`-`"
- "`==`", "`=`", "`!=`", "`<>`", "`<`", "`<=`", "`>`", "`>=`"
- "`and`", "`and then`" (not available as constructor), "`or`", "`or else`" (not available as constructor), "`xor`", "`=>`", "`<=>`"

The symbols "`and`", "`or`", "`xor`", "`div`", "`mod`", and "`rem`" are keywords, which must be written using lower-case letters. Identifiers containing upper-case letters (e.g., '`AND`" or "`Div`") are assumed to be user-defined infix operators (with highest precedence). To avoid any confusion with the corresponding lower-case infix operators, a warning is emitted if parentheses are missing. The symbols "`and then`" and "`or else`" are also keywords and using upper-case letters would trigger a syntax error.

The infix Boolean connectors "`and`", "`and then`", "`or`", "`or else`", "`xor`", "`<=>`", and "`=>`" having the same precedence, parentheses should be used when combining them. Absence of parentheses triggers a warning, as for instance "`x and y or z`". Similarly, parentheses should be used when combining distinct infix functions, which are neither keywords nor key symbols (i.e., "other infix operators" in the above list).

All (infix) operators of same precedence are parsed from left to right, meaning that "$V_1$ $op_1$ $V_2$ $op_2$ $V_3$" is parsed as "($V_1$ $op_1$ $V_2$) $op_2$ $V_3$" rather than "$V_1$ $op_1$ ($V_2$ $op_2$ $V_3$)".

Examples follow:

- "`E (-1 == x - 2)`" is parsed as "`E (-1 == (x - 2))`", since "`-`" has precedence over "`==`".
- "`x gcd 1 + y`" is parsed as "`(x gcd 1) + y`", since "`gcd`" has precedence over "`+`".

- "`x gcd y scm z`" triggers a warning, since "`gcd`" and "`scm`" are not keywords and have same precedence.

To ensure the compatibility between LNT2LOTOS and TRAIAN, the first symbol following $T$ in a pattern of the form "**any** $T$" should not be an identifier. Otherwise, a warning message is triggered. The warning can be suppressed by using parentheses, e.g., a pattern of the form "**any** $T$ $C$ $P$" should rather be written "**(any** $T$**)** $CP$". Note that "**any** $T$ **of** $T$" does not require parentheses around "**any** $T$", as "**of**" is a keyword.

## 7.4 Variables

The data part of LNT is a fully imperative language in syntax and semantics.

LNT supposes the existence of a memory: a set of variables (noted $X$ in this manual) which can store values, and which can be accessed for read and write operations.

However, the static semantics constraints impose a clean imperative style, in the sense that errors in manipulation of variables are signalled at compile-time, and should not produce runtime errors.

These static semantics constraints are based on two principles:

(VAR1) LNT is strongly typed: each variable $X$ must be *declared* before being *used*. The declaration assigns $X$ a type $T$, and $X$ keeps the same type $T$ throughout its *lifetime*.

Variables are *declared* in "**var** ... **end var**" statements, "**case** ... **end case**" statements, and function definitions (as formal parameters).

Variables are *used* in value expressions and function calls.

The *lifetime* (or scope) of a variable extends from its declaration to the end of the statement in which it is declared (for "**var** ... **end var**" and "**case** ... **end case**" statements), or in the whole function definition (for formal parameters of functions). Outside this scope, the variable does not exist. Declarations can be nested: any re-declaration, whether with the same type or a different type, hides the outer declaration.

Variables and expressions are strongly typed.

(VAR2) Access to an uninitialized variable is signalled at compile-time: variables must be *assigned* before being *used*.

Variables can be *assigned* in assignment statements "$X := V$", by procedure calls with "**out**", "**out var**", or "**in out**" parameters, or by patterns in case statements.

A consequence of this constraint is that every "**out**" or "**out var**" function parameter must be assigned before the function returns.

## 7.5 Function definitions

A function definition consists of a function name $F$, optional $\mathit{formal\_events}_0$**,**...**,**$\mathit{formal\_events}_m$, a (possibly empty) list of formal parameters $\mathit{formal\_parameter}_1$**,**...**,**$\mathit{formal\_parameter}_n$, an optional return type $T$, optional pragmas $\mathit{function\_pragma}_1$...$\mathit{function\_pragma}_l$, optional preconditions $\mathit{precondition}_1$**,**...**,**$\mathit{precondition}_j$, optional postconditions $\mathit{postcondition}_1$**,**...**,**$\mathit{postcondition}_k$, and some instruction $I_0$ called the body of the function:

*function_definition* ::= **function** $F$ [ [*formal_events*$_0$ ,..., *formal_events*$_m$] ]
[ (*formal_parameters*$_1$ ,..., *formal_parameters*$_n$) ] [ :$T$ ] **is**
*function_pragma*$_1$...*function_pragma*$_l$
*precondition*$_1$...*precondition*$_j$
*postcondition*$_1$...*postcondition*$_k$
[ $I_0$ ]
**end function**

A function can be defined without parameters. In this case, the parentheses can be omitted.

The body $I_0$ computes the result value of $F$ and the output parameters (those declared of mode "**out**", "**out var**", or "**in out**").

The following static semantics constraints apply to $F$:

(FD1) If $F$ has a return type $T$, this type must refer to an existing type.

(FD2) If $F$ has a return type $T$, $I_0$ must return a result of type $T$.

(FD3) Two functions can have the same name if their profiles (*i.e.* the types and modes of formal parameters or the result type) differ. Such functions are said to be *overloaded*.

(FD4) *This constraint was removed in January 2017; see item* `#2276` *in the* `$CADP/HISTORY` *file.*

(FD5) If $F$ has no return type, $I_0$ must not return a result, and must have at least one "**out**", "**out var**", or "**in out**" parameter. Indeed, a procedure with no result and only "**in**" and/or "**in var**" parameters does not perform useful computation. [checked by LNT2LOTOS]

(FD6) If $F$ has a return type $T$ and exactly two parameters, which are "**in**" and/or "**in var**" parameters, then it can be used both in prefix and infix forms.

(FD7) If the name of $F$ is a special identifier, then $F$ must have a result type.

(FD8) If the name of $F$ is a special identifier, then $F$ must not have any "**out**", "**out var**", or "**in out**" parameter.

## 7.6 Function pragmas

The optional pragmas attached to a function give hints about how the translation to LOTOS and C of the source code should be performed.

When translating a non-external LNT function $F$, LNT2LOTOS may generate one or several LOTOS operations. More precisely, there will be one LOTOS function generated to compute the result of $F$, plus one LOTOS function for each "**out**", "**out var**", and "**in out**" parameter of $F$.

The "`!implementedby "LOTOS:..."`" pragma attached to an LNT function $F$ enables one to specify precise names to be used by LNT2LOTOS when generating the LOTOS function(s) corresponding to the translation of $F$. This pragma is useful when interfacing generated LOTOS code with hand-written LOTOS code.

The name of the generated LOTOS functions is determined by *name* within the "`!implementedby "LOTOS:`*name*`"`" pragma. Precisely, if $F$ has no "**out**"/"**out var**"/"**in out**" parameter, then a single LOTOS function is generated. The name of this LOTOS function is *name*. Otherwise, the name of each LOTOS function is determined by concatenating *name* with two consecutive underscore characters ("__") and the name of the corresponding "**out**"/"**out var**"/"**in out**" parameter of $F$, or with the special name "return" to denote the result returned by $F$.

The "`!implementedby "C:`*name*`"`" (or simply "`!implementedby "`*name*`"`") pragma attached to an LNT function $F$ enables one to specify precise names to be used by CÆSAR.ADT when generating the C function(s) corresponding to the translation of the LOTOS operation(s) generated by LNT2LOTOS for $F$. This pragma is useful when interfacing generated LOTOS code with hand-written or external C code. The C name is determined by *name* using the same rules as for the "`!implementedby "LOTOS:`*name*`"`" pragma.

The "`!external`" pragma attached to an LNT function $F$ enables one to use external (handwritten) C or LOTOS functions in an LNT module. Precisely, if $F$ has an "`!external`" pragma, LNT2LOTOS behaves as follows:

- If $F$ has an "`!implementedby "LOTOS:`*name*`"`" pragma, then LNT2LOTOS does not generate any LOTOS function corresponding to $F$ as this function is supposed to be defined by external LOTOS code.

  Note that *name* is currently silently ignored; the external function is supposed to have the same name as $F$. This limitation will be removed in the future.

  Note also that if $F$ has "**require**" or "**ensure**" clauses, then these clauses are ignored. This limitation will be removed in the future.

- Otherwise, LNT2LOTOS generates LOTOS functions named in the same way as non-external functions but equipped with a special LOTOS comment that declares them as external C functions in the generated LOTOS code. The corresponding C functions must be be provided in a "`.fnt`" file (see Section 2.5).

The following static semantics constraints apply to the pragmas of a function $F$:

(FPG1) In the list *function_pragma*$_1$...*function_pragma*$_n$ of each function definition, there should be at most one pragma of each kind (i.e., there cannot be two "`!external`" pragmas, nor two "`!implementedby "LOTOS:..."`" pragmas, nor two "`!implementedby "..."`"/"`!implementedby "C:..."`" pragmas, etc.)

(FPG2) If the pragma "`!external`" is present, the body $I_0$ should be "**null**".

(FPG3) If both pragmas "`!external`" and "`!implementedby "LOTOS:..."`" are present, then the function can neither have "**out**", "**out var**", nor "**in out**" parameters. [checked by LNT2LOTOS]

(FPG4) In pragmas "`!implementedby "`*name*`"`" and "`!implementedby C:`*name*`"`" for all functions, the values of *name* should be pairwise distinct.

(FPG5) *This constraint was removed in September 2022; see item* `#2822` *in the* `$CADP/HISTORY` *file.*

(FPG6) *This constraint was removed in December 2022; see item* `#2837` *in the* `$CADP/HISTORY` *file.*

(FPG7) To avoid name clashes in the generated LOTOS code, the name provided by a pragma "`!implementedby "LOTOS:`*name*`"`" to a non-external function should not be a name of another LNT function, either predefined or defined by the user. In particular, cyclic or self references such as "**function** `F` **is !implementedby** `"LOTOS:F"` ..." are forbidden unless `F` is declared as "`!external`". [checked by CÆSAR/CÆSAR.ADT]

In pragmas "`!implementedby "C:..."`" and "`!implementedby "LOTOS:..."`", The prefixes "`C:`" and "`LOTOS:`" are case-sensitive. Other forms, such as "`c:`" and "`Lotos:`" are rejected.

## 7.7 Lists of formal events

*formal_events* ::= *event_declaration*

The above clause declares a list of formal event parameters $E_0, \ldots, E_n$.

The following static semantic constraints hold:

(FE1) In a function (resp. process) definition, the formal events $E_0, \ldots, E_n$ must be pairwise distinct.

(FE2) In a function (resp. process) definition, each formal event $E_i$ must be different from the predefined event noted "**i**".

(FE3) In a function (resp. process) definition, each formal event $E_i$ must be different from the predefined event noted "**unexpected**".

(FE4) In a function definition, the channel Γ must be equal to "**exit**" (see Section 6.5). [checked by LNT2LOTOS]

## 7.8 Lists of formal parameters

*formal_parameters* ::= *parameter_mode* $X_0, \ldots, X_n : T$

The above clause declares a list of variable parameters $X_1, \ldots, X_n$, which all have the same mode *parameter_mode* and the same type $T$.

(FP1) In a function definition, the names of the formal parameters must be pairwise distinct.

(FP2) $T$ must refer to an existing type.

## 7.9 Modes of formal parameters

*parameter_mode* ::= [ **in** ]
| **in var**
| **out**

| **out var**
| **in out**

A value parameter declared with the keyword "**in**" denotes a constant parameter. The body $I_0$ of the function should not change the value of an "**in**" parameter.

A value parameter declared with the keyword "**in var**" denotes a constant parameter. The body $I_0$ of the function can change the local value of an '**in var**" parameter, but this change is invisible to the caller.

A value parameter declared with the keyword "**out**" is a result parameter that must be assigned by $I_0$, and its value is visible after the function call. The body $I_0$ of the function should not read the value of an "**out**" parameter. Yet, the value of an "**out**" parameter may be read in the postconditions of the function.

A value parameter declared with the keyword "**out var**" is a result parameter that must be assigned by $I_0$, and its value is visible after the function call. The body $I_0$ can read the value of an "**out var**" parameter after it has been assigned. Note however that the value of an "**out**" parameter can be read in a postcondition or in the "**where**" clause of a nondeterministic assignment defining it (as in, e.g., "$x$ **:= any nat where** $x$>0") without having to declare it as "**out var**".

A value parameter declared with the keyword "**in out**" is a modifiable parameter that has an initial value. $I_0$ may modify this value. The value of the parameter assigned by $I_0$ is visible after the function call.

The default mode is "**in**".

The following static semantics constraint applies to the body $I_0$ of function $F$:

(FA1) For each formal parameter $X$ with mode "**out**" or "**out var**", $X$ must be assigned a value on all execution paths before $F$ returns. Section 7.11 explains how a variable can be assigned a value, and how a function can return.

(FA2) For each formal parameter $X$ with mode "**in out**", $X$ must be assigned a value before $F$ returns. Otherwise, $X$ should be rather declared with mode "**in**".

(FA3) For each formal parameter $X$ with mode "**in out**", there should exist at least one execution path on which the value of $X$ is read before $F$ returns and before $X$ is modified again (should it be). Otherwise, $X$ should rather be removed (if $X$ is also never assigned), or declared with mode "**out**".

(FA4) For each formal parameter $X$ with mode "**out var**", there should exist at least one execution path on which the value of $X$ is read after being assigned. Otherwise, $X$ should be rather declared with mode "**out**".

(FA5) For each formal parameter $X$ with mode "**in**", $X$ should never be assigned.

(FA6) For each formal parameter $X$ with mode "**in**" or "**in var**", there should exist at least one execution path on which the value of $X$ is read before $F$ returns and before $X$ is modified again (should it be). Otherwise, $X$ should rather be removed (if $X$ is also never assigned), or transformed into a local variable.

(FA7) For each formal parameter $X$ with mode "**in var**", there should exist at least one execution path on which the value of $X$ is modified. Otherwise, $X$ should be rather declared with mode "**in**".

In the rest of this document, a function that has at least one formal parameter declared with mode "**out**", "**out var**", or "**in out**", is called a *procedure*.

## 7.10 Preconditions and postconditions

*precondition* ::= **require** $V$ [ **raise** $E$ [ `()` ] ];

*postcondition* ::= **ensure** $V$ [ **raise** $E$ [ `()` ] ];

A "**require**" clause denotes a function *precondition*. $V$ is a Boolean expression, whose variables must be formal parameters declared with mode "**in**", "**in var**", or "**in out**" (see Section 7.9).

When entering a function, a precondition "**require** $V$ **raise** $E$" or "**require** $V$ **raise** $E$ `()`" raises exception $E$ if the value of $V$ is false when replacing every parameter $X$ by its input value. The syntax "**require** $V$" is a shorthand notation for "**require** $V$ **raise unexpected**".

An "**ensure**" clause denotes a function *postcondition*. $V$ is a Boolean expression, whose variables must be formal parameters. In $V$, if $X$ is a parameter declared with mode "**in var**" or "**in out**" (see Sections 7.9 and 7.13.1), then the value expression "$X$ **.in**" (resp. "$X$ **.out**") denotes the input (resp. output) value of $X$. $V$ may also contain occurrences of the keyword "**result**" to denote the function result, if any (see Section 7.13.2).

Just before leaving a function normally (i.e., when the control reaches a "**return**" instruction or the end of the function, but not when an exception is raised), a postcondition "**ensure** $V$ **raise** $E$" or "**ensure** $V$ **raise** $E$ `()`" raises exception $E$ if the value of $V$ is false. The syntax "**ensure** $V$" is a shorthand notation for "**ensure** $V$ **raise unexpected**".

(AF1) If event $E$ is different from "**unexpected**", then it must have been declared as a parameter of the current function.

(AF2) The variables occurring in a precondition must be formal parameters declared with mode "**in**", "**in var**", or "**in out**".

(AF3) The variables occurring in a postcondition must be formal parameters.

(AF4) "$X$ **.out**" may occur only in postconditions of functions and processes that declare $X$ as a parameter of mode "**in var**" or "**in out**".

(AF5) "$X$ **.in**" may occur only in postconditions of functions and processes that declare $X$ as a parameter of mode "**in var**" or "**in out**".

(AF6) In a postcondition, if $X$ is a parameter declared with mode "**in var**" or "**in out**", then it can only occur in $V$ under one of the forms "$X$ **.in**" or "$X$ **.out**" (excluding the ambiguous form "$X$"). However, if $X$ is declared with mode "**in var**", then the form "$X$ **.out**" is not recommended and issues a warning.

(AF7) In a postcondition, the "**result**" keyword can be used only if the function returns a value.

Note: A function $F$ declared with the pragma "`!external`" can have preconditions and/or postconditions. In such a case, before translation to LOTOS, LNT2LOTOS expands $F$ into two LNT functions: an auxiliary function with same profile as $F$ but new (unused) name, to which the "`!external`" and

“`!implementedby "..."`”/“`!implementedby "C:..."`” pragmas of $F$ are attached, and whose body is defined as “**null**”; and the function $F$ itself, from which the “`!external`” and “`!implementedby "..."`”/“`!implementedby "C:..."`” pragmas are detached, and whose “**null**” body is replaced by a call to the above auxiliary function. As expected, when $F$ is called from an LNT program, the preconditions (if any) are checked, the external function is called, and finally the postconditions (if any) are checked. Obviously however, calling the external C function directly from a C program does not yield a verification of the preconditions and/or postconditions. Note that this expansion of $F$ does not change the name of the LOTOS operations implementing $F$. In particular, if $F$ has an “`!implementedby "LOTOS:..."`” pragma, then this pragma remains attached to $F$. Therefore, calling these operations from a LOTOS program remains possible in the same way as any other function.

Note: Each semicolon occurring between successive preconditions (resp. postconditions) is a sequential composition operator, meaning that their execution is sequential. If several preconditions (resp. postconditions) are violated, the exception corresponding to the first one (in reading order) is raised. By contrast, the semicolon delimiting the sequence of preconditions and the sequence of postconditions and the semicolon delimiting the sequence of postconditions and the function body are not sequential composition operators, since the postconditions are not evaluated where they appear, but only when leaving the function.

## 7.11 Statements

Each LNT statement is expected to terminate. Although termination cannot be checked automatically in the general case, LNT2LOTOS may issue error messages when it is certain that a given statement (e.g., the body of a function or a procedure) will never terminate.

### 7.11.1 Null statement

This statement has no effect. It does not return any value nor assign any variable.

### 7.11.2 Sequential composition

The evaluation of the sequential composition “$I_1$ ; $I_2$” starts by evaluating $I_1$, and then evaluating $I_2$.

### 7.11.3 Return statement

In its simplest form, without a value expression, “**return**” makes the function return.

“**return** $V$” evaluates the expression $V$ and makes the function return the value of $V$.

(R1) The simple “**return**” form can be used if and only if the function has no return type.

(R2) If the function has a return type, each execution path must contain a “**return** $V$” statement.

(R3) In “**return** $V$”, the type of $V$ must be the return type of the function.

### 7.11.4 Exception raise

"**raise** $E$" makes the currently executed program print a message and stop.

For debugging purpose, this message includes the name of the identifier $E$ (if different from "**unexpected**") and the name of the current LNT function (resp. process).

(ER1) If event $E$ is different from "**unexpected**", then it must have been declared by the function (resp. the process) that contains the "**raise**" statement.

(ER2) $E$ must be an exception, which implies that its channel is "**exit**". [checked by LNT2LOTOS]

Note: The current LNT syntax, i.e., "**raise** $E$" or "**raise** $E$ `()`", enforces the restriction that exceptions do not carry value parameters (see Section 6.5).

### 7.11.5 Assertion

An assertion statement "**assert** $V$ **raise** $E$" or "**assert** $V$ **raise** $E$ `()`" raises exception $E$ if the value of $V$ is false. Otherwise, it is equivalent to **null**. This statement is thus equivalent to "**if** $V$ **then null else raise** $E$".

The syntax "**assert** $V$" is a shorthand notation for "**assert** $V$ **raise unexpected**".

The constraints (ER1) and (ER2) of Section 7.11.4 also apply to assertion statements.

Note that "**assert** $V$ **raise** $E$" and "**assert** $V$ **; raise** $E$" are fundamentally different, even if they only differ by the presence of a semicolon. In the latter case, an exception (either $E$ or "**unexpected**") will always be raised, whatever the value of $V$.

### 7.11.6 Array element assignment

The statement "$X$ `[` $V_0$ `]` `:=` $V_1$" modifies the value stored at index $V_0$ of the variable $X$.

Note that neither $V_0$ nor $V_1$ can contain procedure calls, *i.e.* calls to function with "**out**", "**out var**", or "**in out**" parameters.

(SAA1) The type $T$ of variable $X$ must be an **array** type.

(SAA2) Expression $V_0$ must be of type $NAT$.

(SAA3) Expression $V_1$ must have the same type as elements of array type $T$.

Note that before being able to use $X$ in an expression, $X$ must have been assigned a value with a statement of the form "$X$ `:=` $V$". Therefore, initializing each element of array $X$ with a statement of the form "$X$ `[` $V_0$ `]` `:=` $V_1$" is not sufficient to initialize $X$. This is because the LNT2LOTOS translator cannot statically ensure that all the elements are initialized.

### 7.11.7 Procedure call

A procedure call has the form "[ **eval** ] [ $X$ `:=` ] $F$ [ `[` *actual_events* `]` ] `(`*actual_parameter*$_1$`,` ..., *actual_parameter*$_n$`)`".

The **eval** keyword is optional in function definitions, whereas it is mandatory in process definitions if the "$X$ `:=`" part is absent (i.e., the function does not return a result); if not, there would be an ambiguity between procedure calls and communications in the syntax of behaviours (see Section 8.2).

When $F$ is called, its formal event parameters (if any) are replaced with the corresponding actual event parameters according to the standard *call-by-value* semantics

The actual events can be written either in the "positional" style or in the "named" one. In the named style:

- The notation "$E_{formal,i}$ `->` $E_{actual,i}$" means that the formal event parameter $E_{formal,i}$ of function $F$ is instantiated with the actual event $E_{actual,i}$.
- The notation "`...`" means that each formal event parameter $E$ of $F$ that does not appear in $E_{formal,1}$, ..., $E_{formal,n}$ is instantiated with the actual event $E$.

The static semantics constraints (PE1) and (PE2) apply to the positional style "$E_1$`,`...`,`$E_n$". The static semantics constraints (PE3) to (PE8) apply to the named style "$E_{formal,1}$ `->` $E_{actual,1}$`,`...`,`$E_{formal,n}$ `->` $E_{actual,n}$[`,...`]". The remaining constraints (PE9) and (PE10) apply to both positional and named style.

(PE1) The number of actual event parameters of the procedure call must be equal to the number of formal event parameters of the corresponding procedure definition.

(PE2) Each actual event parameter $E_1, ..., E_n$ must have been declared in the current context, i.e., in the function (resp. the process) that contains the call to $F$, except for the predefined event "**unexpected**".

(PE3) The formal events $E_{formal,1}$, ..., $E_{formal,n}$ must be formal events of $F$ and be pairwise distinct.

(PE4) Each actual event parameter $E_{actual,1}, ..., E_{actual,n}$ must have been declared in the current context, i.e., in the function (resp. the process) that contains the call to $F$, except for the predefined event "**unexpected**".

(PE5) If the notation "`...`" is used in "$E_{formal,1}$ `->` $E_{actual,1}$`,`...`,`$E_{formal,n}$ `->` $E_{actual,n}$`,...`", all the formal events of $F$ that do not appear in $E_{formal,1}$, ..., $E_{formal,n}$ must have been declared in the function (resp. the process) that contains the call to $F$.

(PE6) When "`...`" is omitted, all the formal events of $F$ must appear in $E_{formal,1}$, ..., $E_{formal,n}$.

(PE7) If event parameters are passed to $F$ in the named style, and if $F$ has several overloaded definitions in the current module, then each of those definitions of $F$ must have the same formal event parameters, in the same order.

(PE8) Function $F$ must be defined in the current module, meaning that the named style can only be used to call routines in the same module (because, at present, Lnt2Lotos does not do sophisticated inter-module analysis). [checked by Lnt2Lotos]

(PE9) If $F$ has event parameters but does not have value parameters, then empty parentheses are mandatory when calling this procedure, as in "$F$`[`$E_0$`,`...`,`$E_n$`] ()`", so as to avoid syntactic ambiguity with array elements when $m = 0$.

(PE10) In either named or positional style, each actual event must be compatible (as defined in Section 6.7) with the corresponding formal event of $F$.

Each actual parameter corresponds to a formal parameter in the definition of the procedure $F$ (see Section 7.9) in the usual way. An actual parameter is one of the following:

- An expression $V$ denotes an actual parameter corresponding to an "**in**" or "**in var**" formal parameter. Its value is just passed to the procedure and its variables remain unchanged when the procedure returns.
- A variable prefixed with a question mark, as in "`?`$X$", denotes an actual parameter corresponding to an "**out**" or "**out var**" formal parameter. The variable $X$ does not need to have a value before the procedure call. It is necessarily assigned a value when the procedure returns.
- A variable prefixed with an exclamation mark and question mark, as in "`!?`$X$", denotes an actual parameter corresponding to an "**in out**" formal parameter. The variable $X$ must have a value before the procedure call. This value is passed to the procedure, and the variable $X$ may be assigned a new value when the procedure returns.

(PC1) The number of actual parameters of the procedure call must be equal to the number of formal parameters of the corresponding procedure definition.

(PC2) Each expression $V$ must have the same type and must appear at the same position as the corresponding "**in**" or "**in var**" parameter of the procedure definition.

(PC3) Each variable "`?`$X$" must have the same type and must appear at the same position as the corresponding "**out**" or "out var" parameter of the procedure definition.

(PC4) Each variable "`!?`$X$" must have the same type and must appear at the same position as the corresponding "**in out**" parameter of the procedure definition.

(PC5) "$X$ :=" is present if and only if $F$ has a return type. In this case, the type of $X$ must be the same as the return type of $F$.

(PC6) If the procedure $F$ is overloaded, the information given by the types and modes of its parameters and the type of the resulting value should suffice to solve the overloading.

(PC7) The "**out**", "**out var**", and "**in out**" parameters should be pairwise distinct (*i.e.* "$F($`?`$X,$`?`$X)$" is forbidden).

(PC8) If the "[ **eval** ] $X := F$ `(` *actual_parameter*$_1$ `,` ... `,` *actual_parameter*$_n$ `)` " form is used then $X$ cannot appear among more than once in the "**out**", "**out var**", and "**in out**" parameters (*i.e.* "$X := F($`?`$X)$" is forbidden).

(PC9) The assignment and "**out**", "**out var**", or "**in out**" parameter passing should be "useful", i.e., for each variable $X$ occurring on the left-hand side of the assignment symbol ":=", or passed as an actual parameter ("`?`$X$" or "`!?`$X$"), there should exist at least one execution path on which the new value assigned to $X$ is read before the execution completes and before $X$ is modified again (should it be).

The evaluation of a procedure call begins with the simultaneous evaluation of expressions corresponding to the "**in**" and "**in var**" parameters. For the "**in out**" parameters, the input value is the value of the variable given as a parameter. Then, the body of the procedure is evaluated in the context of actual values for "**in**", "**in var**", and "**in out**" parameters. The body should assign all the "**out**" and "**out var**" parameters and should return a value if $F$ returns a value.

### 7.11.8 Variable declaration

The following statement:

**var**
  *var_declaration*$_0$ **,**
  ... **,**
  *var_declaration*$_n$
**in**
  $I_0$
**end var**

declares local variables: their names and their types.

The scope of each variable is $I_0$.

Scoping is lexical: any re-declaration of a variable hides the outer declaration.

The declaration must obey the following rule:

(VD1) The names of the variables declared in "*var_declaration*$_0$ **,** ... **,** *var_declaration*$_n$" must be pairwise distinct.

### 7.11.9 Case statement

The most general conditional statement offered by LNT is the "**case**" statement:

**case** $V_0$ **,** ... **,** $V_\ell$
  [ **var** *var_declaration*$_0$ **,** ... **,** *var_declaration*$_n$] **in**
    *match_clause*$_0$ `->` $I_0$
   `|` ...
   `|` *match_clause*$_n$ `->` $I_n$
**end case**

First, the expressions $V_0$, ..., $V_\ell$ are evaluated. Then, the statement $I_i$ corresponding to the first *match_clause_i* that matches $V_0$, ..., $V_\ell$ is executed.

Optionally, some variables can be declared at the beginning of the "**case**" statement. Their scopes are the match clauses and the statements $I_0$, ..., $I_n$.

The case patterns in the match clauses *match_clause*$_0$, ..., *match_clause*$_n$ can bind variables declared in the optional variable declaration, as well as previously declared variables. The scope of a variable binding by a case pattern is limited by the variable's declaration only.

For example, function "`f`" modifies its local variable $y$ either in the "`Succ (`$y$`)`" pattern or in the "$y$ `:= 0`" assignment.

**function** `f` (`x` : `Nat`) : `Nat` **is**

```
    var y : Nat in
        case x in
            Succ (y) -> null
        | any -> y := 0
        end case;
        return y
    end var
end function
```

For example, the function "decr" decrements its parameter $x$ by one if $x$ is not already zero. The variable $x$ is reassigned by the case statement and keeps its value until the end of the function.

```
function decr (in var x : Nat) : Nat is
    case x in
        Succ (x) -> null
    | any -> null
    end case;
    return x
end function
```

(CS1) The clauses *match_clause*$_0$, ..., *match_clause*$_n$ must be exhaustive: they must cover all the possible sequences of values of the sequence of the types of $V_0$, ..., $V_\ell$. This ensures that exactly one statement among $I_0$, ..., $I_n$ will be executed at runtime. This is checked at compile-time by TRAIAN; moreover, the LOTOS and C code generated by LNT2LOTOS is such that case statements that are not exhaustive will abort at run-time when executed with a value $V$ that is not covered by the clauses *match_clause*$_0$, ..., *match_clause*$_n$.]

(CS2) The variables bound in the "**case**" pattern should be used afterwards; see rule (PA4).

Due to typing limitations in the early versions of the LNT2LOTOS translator, each "$V_i$" should either be a variable "$X$" or an expression of the form "$V_i$ **of** $T$" ($\forall i \in \{0, ..., \ell\}$). This constraint may be relaxed in the future.

A *match_clause* has the form "$P_{(0,0)}$ , ..., $P_{(0,\ell)}$ [**where** $V_0$] | ... | $P_{(k,0)}$ , ..., $P_{(k,\ell)}$ [**where** $V_k$]". Semantically, "*match_clause* `->` $I$" is strictly equivalent to:

$P_{(0,0)}$ , ..., $P_{(0,\ell)}$ [ **where** $V_0$ ] `->` $I$
| ...
| $P_{(k,0)}$ , ..., $P_{(k,\ell)}$ [ **where** $V_k$ ] `->` $I$

The number $\ell$ of patterns must be equal to number of expressions in the **case** statement, and the corresponding types have to match.

A sequence of expressions is said to match a clause "$P_0$ , ..., $P_\ell$ **where** $V$" when:

- it first matches the sequence of patterns $P_0$ , ..., $P_\ell$, and then
- the evaluation of $V$ in the context of variables bound by the matching returns the Boolean value **true**.

(WT1) The expressions $V_0$, ..., $V_k$ must be of type Boolean.

In "$P_0$**,** ...**,**$P_\ell$ [ **where** ] $V$", each $P_i$ can be equal to "**any**". This possibility is an exception to the rule that wildcards inside patterns have to be typed explicitly (using the notation "**any** T").

Patterns are discussed in section 7.12

### 7.11.10 If statement

The "**if**" construct allows conditional computations, as it is included in all languages.

In LNT it has the following form:

 **if** $V_0$ **then** $I_0$<br>
[ **elsif** $V_1$ **then** $I_1$<br>
  ...<br>
  **elsif** $V_n$ **then** $I_n$ ]<br>
[ **else** $I_{n+1}$ ]<br>
**end if**

(IF1) The expressions $V_0$, ..., $V_n$ must be of type Boolean.

### 7.11.11 Breakable loop statement

A loop statement "**loop** $L$ **in** $I_0$ **end loop**" can be interrupted with the statement "**break** $L$".

(BL1) A "**loop** $L$" statement cannot be declared inside a loop statement which has the same label $L$.

(BL2) A "**loop** $L$" statement must contain either a "**return**" statement, a "**break** $L$" statement, or a "**break** $L'$" statement such that the "**loop** $L$" statement occurs inside "**loop** $L'$" (otherwise it would be certain that the "**loop** $L$" never terminates).

### 7.11.12 Unbreakable loop statement

An unbreakable loop statement "**loop** $I_0$ **end loop**" cannot be interrupted with a "**break** $L$" statement. However, since infinite computations should be avoided in the data part, the following static semantic constraint should be satisfied:

(UL1) A "**loop**" statement must contain either a "**return**" statement or a "**break** $L$" statement such that the "**loop**" statement occurs inside "**loop** $L$" (otherwise it would be certain that the "**loop**" never terminates).

### 7.11.13 Breakable while statement

A loop statement "**while** $V$ **loop** $L$ **in** $I_0$ **end loop**" can be interrupted with the statement "**break** $L$".

(BW1) The expression $V$ must be of type Boolean.

(BW2) A "**while** ... **loop** $L$" statement cannot be declared inside a loop statement which has the same label $L$.

### 7.11.14 Unbreakable while statement

An unbreakable while statement "**while** $V$ **loop** $I_0$ **end loop**" cannot be interrupted with a "**break** $L$" statement.

(UW1) The expression $V$ must be of type Boolean.

### 7.11.15 Breakable for statement

A loop statement "**for** $I_0$ **while** $V$ **by** $I_1$ **loop** $L$ **in** $I_2$ **end loop**" can be interrupted with the statement "**break** $L$".

It is semantically equivalent to:

$I_0$ ;
**loop** $L$ **in**
  **if** $V$ **then** $I_2$ ; $I_1$
  **else break** $L$
  **end if**
**end loop**

(BF1) The expression $V$ must be of type Boolean.

(BF2) A "**for** ... **loop** $L$" statement cannot be declared inside a loop statement which has the same label $L$.

### 7.11.16 Unbreakable for statement

A loop statement "**for** $I_0$ **while** $V$ **by** $I_1$ **loop** $I_2$ **end loop**" cannot be interrupted with a "**break** $L$" statement.

This statement is similar to the "**for**" construct of the C language. It is semantically equivalent to the statement defined above for breakable "**for**" statements, except that $L$ must be a loop label that is not used elsewhere.

(UF1) The expression $V$ must be of type Boolean.

### 7.11.17 Break statement

The "**break** $L$" statement can be used to interrupt a breakable "**loop**", "**while**", or "**for** statement".

Loop statements can be nested, and the "**break**" statement enables one to interrupt a loop which is not the innermost loop.

(BS1) A statement "**break** $L$" can only be used within the body $I$ of a loop statement of either form "**loop** $L$ **in** $I$ **end loop**", "**while** ... **loop** $L$ **in** $I$ **end loop**", or "**for** ... **loop** $L$ **in** $I$ **end loop**" declared with the same label $L$.

### 7.11.18 Use statement

The statement "**use** $X_1$**,** ...**,** $X_n$" marks the variables $X_1, ..., X_n$ as used as if they were used in a normal expression. Apart from this, it has the same semantics as **null**.

(U1) The variables $X_1, ..., X_n$ must have been assigned previously.

This statement is useful to eliminate warnings about unused "**in**", "**in var**", and "**in out**" parameters when the parameter modes cannot be changed and the parameters cannot be removed; see rules (FA3) and (FA6) page 63.

This statement could also be used to eliminate warnings about local variables that are not used after being assigned. However in this case, it is recommended to rather remove the useless variable assignments instead of resorting to unnecessary "**use**" statements.

In general, one should avoid writing "**use**" statements that are not strictly necessary. Thus, it is forbidden to introduce a "**use** $X$" on an execution path where variable $X$ has been already used or will be used. At present, such check is not yet implemented, but it should be in the future.

### 7.11.19 Access statement

The statement "**access** $E_1$**,** ...**,** $E_n$" marks the events $E_1, ..., E_n$ as accessed as if they occurred in a **raise** statement (in a function or in a process) or in a communication action (in a process only, see Chapter 8). Apart from this, it has the same semantics as **null**.

(A1) In a function (resp. process) definition, each formal event $E_i$ must be different from the predefined event noted "**i**".

(A2) In a function (resp. process) definition, each formal event $E_i$ must be different from the predefined event noted "**unexpected**".

This statement is useful to eliminate warnings about non-accessed events.

In general, one should avoid writing "**access**" statements that are not strictly necessary. Thus, it is forbidden to introduce a "**access** $E$" on an execution path where event $E$ has been already accessed or will be accessed. At present, such check is not yet implemented, but it should be in the future.

## 7.12 Patterns

### 7.12.1 Variable binding

The variables $X$ belonging to a pattern $P$ are "initialization" occurrences: they previously have been declared, but do not have to be initialized.

(PA1) The same variable $X$ cannot be used more than once in the pattern sequence $P_0$, ..., $P_\ell$ of a single match-clause.

(PA2) The patterns $P_0, \ldots, P_n$ in a constant pattern $F(P_0, \ldots, P_n)$ or in the infix constant pattern $P_1\ F\ P_2$, cannot contain any variable, wildcard, or aliasing patterns.

(PA3) If a constant pattern of type $T$ is used, an equality operator "`==`" of type $T, T \rightarrow$ `Bool` must be in scope in the same module. [checked by CÆSAR]

(PA4) For each variable $X$ bound by the pattern, there should exist at least one execution path on which the new value of $X$ is read before the execution completes and before $X$ is modified again (should it be). Otherwise, "**any**" should be used in place of $X$ in the pattern.

### 7.12.2 Pattern matching

The pattern-matching of a value $V$ with a pattern $P$ has two effects:

- It sends a Boolean result which is "**true**" if $V$ has the same structure as $P$ (i.e., $V$ matches $P$), or "**false**" otherwise.

- If $V$ matches $P$, the variables $X$ used by $P$ are initialized with the values extracted from $V$.

The result returned by matching a list of a list of values "$V_0$ `,` ...`,`$V_\ell$" with a list of patterns "$P_0$ `,` ...`,`$P_\ell$" is the conjunction of the results returned by matching $V_i$ with $P_i$.

Matching is recursively defined as follows:

| Pattern | Value | Condition | Effect | Result |
|---|---|---|---|---|
| $X$ | $V$ | None | $X$ receives $V$ | true |
| **any** $T$ | $V$ | None | None | true |
| $X$ **as** $P$ | $V$ | $P$ and $V$ match | $X$ receives $V$ | true |
| $X$ **as** $P$ | $V$ | $P$ and $V$ do not match | None | false |
| $C(P_0, \ldots, P_n)$ | $C(V_0, \ldots, V_n)$ | Each $P_i$, $V_i$ match | None | true |
| $F(P_0, \ldots, P_n)$ | $V$ | $F(P_0, \ldots, P_n)$ equals $V$ according to the operator `==` | None | true |
| $F(P_0, \ldots, P_n)$ | $V$ | $F(P_0, \ldots, P_n)$ does not equal $V$ according to the operator `==` | None | false |
| $C(P_0, \ldots, P_n)$ | $C(V_0, \ldots, V_n)$ | Some $P_i$, $V_i$ do not match | None | false |
| $C(P_0, \ldots, P_n)$ | $V$ | $V$ has not the form $C(V_0, \ldots, V_n)$ or some $V_i$ has not the same type as $P_i$ | None | false |
| $P$ **of** $T$ | $V$ | Same as matching $P$ and $V$ | | |
| $P_1\ C\ P_2$ | $V$ | Same as matching $C(P_1, P_2)$ and $V$ | | |
| $P_1\ F\ P_2$ | $V$ | Same as matching $F(P_1, P_2)$ and $V$ | | |

Note that the pattern "$P_1\ C\ P_2$" has the same meaning as "$C$ `(`$P_1$ `,` $P_2$`)`" when the infix notation of constructor $C$ is used.

Note that the pattern "`(`$P$`)`" has the same meaning as "$P$", but is required for instance to express right associativity of infix constructors.

Constant patterns are compared using the operator "`==`". It is the responsibility of the user to provide a sensible implementation of that operator.

### 7.12.3 List patterns

A list pattern of the form "`{P1, P2, ..., Pn}`" is syntactically replaced by "`cons (P1, cons (P2, ... cons (Pn, nil)...))`". For example, `{}` is converted to `nil` and `{0}` is converted to `cons (0, nil)`.

It is worth noticing that this syntax notation is not restricted to list types. It can also be used for sorted lists, sets, and even any LNT type that has `nil` and `cons` operations with the right profiles. Note that `nil` and `cons` do not need to be constructors. Precisely, the following constraints must be satisfied:

(PL1) All the elements P$i$ must be of the same type $T'$ and this type must be declared.

(PL2) The type $T$ of the list pattern result must be declared.

(PL3) The `nil` function must be declared (with the profile specified in section 5.7).

(PL4) The `cons` function must be declared (with the profile specified in section 5.7). If `cons` is not a constructor, then the list elements `P1, ..., Pn` cannot contain any variable, wildcard, or aliasing patterns.

Remind that, in the case of set and sorted list types, and unlike list expressions (see Section 7.13.9), the pattern must take into account the ordering and/or unicity constraints set by its type. For instance, considering a type of either form `set of nat` or `sorted list of nat`, the pattern `{2, 1}` (which is represented internally as `cons (2, cons (1, nil))` cannot be matched by any value, including the value `{2, 1}` (which is represented internally as `cons (1, cons (2, nil))`) since $2 > 1$.

## 7.13 Value expressions

### 7.13.1 Variable

A value expression may be a variable $X$. The type of the expression is the type of the variable, and the result of the expression evaluation is the value of the variable $X$.

Only in postconditions of functions and processes, variables declared as "**in out**" formal parameters may only occur in one of the forms "$X$ **.in**" or "$X$ **.out**", to distinguish between their input and output values (see Section 7.10).

(EV1) The variable $X$ must have been declared and assigned a value before it is used in an expression.

(EV2) The value expressions "$X$ **.in**" and "$X$ **.out**" can be used only in postconditions.

### 7.13.2 Result

The "**result**" keyword may be used in postconditions of functions that return a result (see Section 7.10).

(RV1) The "**result**" keyword can be used only in postconditions.

### 7.13.3 Constructor call

The constructor call "$C$ [`(`$V_1$`,` ...`,` $V_n$`)`]" computes a value of the domain of its target type.

The infix notation "$V_1$ $C$ $V_2$" is equivalent to "$C$`(`$V_1$`,` $V_2$`)`".

The evaluation of a constructor call begins with the simultaneous evaluation of its actual parameters $V_1$, ..., $V_n$. The values obtained are used to form the constructed value which is the result of the evaluation.

(EC1) Each expression $V$ must have the same type and must appear at the same position as the corresponding parameter of the constructor definition.

(EC2) If the constructor $C$ is overloaded, the information given by the type of its parameters and the type of the resulting value should suffice to solve the overloading.

The *type coercion* operator explained below may help to solve the overloading.

### 7.13.4 Function call

A function call has either form "$F$ [ `[` *actual_events* `]` ] [`(`*actual_parameter*$_1$`,`...`,`*actual_parameter*$_n$`)`]" (prefix notation) or "$V_1$ $F$ [ `[` *actual_events* `]` ] $V_2$" (infix notation). The latter is equivalent to "$F$ [ `[` *actual_events* `]` ] $(V_1, V_2)$".

Function calls allowed inside value expressions are a particular case of procedure calls. Therefore, all constraints that have been defined for procedure calls in Section 7.11.7 also hold for function calls in the context of value expressions. The only additional constraints here are the following:

(FC1) Function $F$ should return a value and cannot have side-effects, *i.e.*, $F$ must have only "**in**" and/or "**in var**" parameters.

(FC2) If $F$ has event parameters and is the left-hand operand of an array element access expression, then parentheses are mandatory around the call to $F$, as in "`(`$F$ `[`$E_0$`,` ...`,` $E_n$`]` `(`$V_1$`,` ...`,` $V_m$`))[`$V'$`]`".

All static semantics constraints given above for constructor calls also apply to function calls.

If $F$ is a LOTOS function, it is the programmer's responsibility to ensure that $F$ has no side effect. A typical LOTOS function cannot have side effects, but a LOTOS function declared `external` can have side effects, depending on its actual `C` implementation.

### 7.13.5 Field selection

A field selection has the form "$V$`.`[ `[`$E$`]` ] *field*" where $V$ is an expression of type $T$, $E$ (optional) is an exception, and *field* is the name of a formal parameter of a constructor of type $T$.

(FS1) At runtime, the value of $V$ must be of the form "$C$ `(`...`)`" where $C$ is a constructor of type $T$ that has a formal parameter named *field*. If not, then an exception ($E$ if it is present or "**unexpected**" otherwise) is raised. [checked at runtime]

The selection expression returns the value of the actual parameter corresponding to *field*.

It is interesting to note that, while field selection is sometimes useful, in most cases it is more efficient to use a "**case**" instruction with pattern matching. Field selection should be used for accessing only one field, whereas pattern matching is better when several fields have to be accessed.

### 7.13.6 Field update

A field update has the form "$V$ `.[` `[`$E$`]` `]` `{`$field_0$ `->` $V_0$`,` ...`,` $field_n$ `->` $V_n$`}`" where $V$, (resp. $E$, $field_0$, ..., $field_n$) respects the same semantic constraints as $V$ (resp. $E$, *field*) in a field selection expression. Additionally:

(FU1) The expressions $V_0$, ..., $V_n$ must have the same type as the corresponding formal parameters $field_0$, ..., $field_n$.

The update expression returns the value of $V$ where the fields $field_0$, ..., $field_n$ have been replaced by the values resulting from the evaluation of the expressions $V_0$, ..., $V_n$.

### 7.13.7 Array element access

An array element access has the form "$V_0$ `[` $V_1$ `]`".

(AE1) The type of expression $V_0$ must be an **array** type.

(AE2) The expression $V_1$ must be of type $NAT$.

(AE3) The value of expression $V_1$ must be a valid index for the array represented by expression $V_0$. [checked at runtime]

### 7.13.8 Type coercion

Type coercion "$V$ **of** $T$" is allowed, to help solve the type ambiguity introduced by function and constructor overloading.

(TC1) $T$ must be a possible type for expression $V$.

Another source of ambiguity is the precedence of infix functions. This precedence can be forced using parenthesized expressions "`(`$V$`)`".

Type coercion also serves to ease the use of range and predicate types (see Section 5.7). If type $T'$ is defined as a range type (i.e., "**type** $T'$ **is range** ... **of** $T$") or as a predicate type (i.e., "**type** $T'$ **is** $\{X : T$ **where** ... $\}$"), and if $V$ is a value of type $T$, it is permitted to write "$V$ **of** $T'$" — in addition to writing "$V$ **of** $T$", which is already permitted for any type $T$. The notation "$V$ **of** $T'$", which enforces the principle of uniform reference, is actually translated to "$T'$ $(V)$ **of** $T'$", where, depending whether $V$ has type $T$ or $T'$, the overloaded function $T'$ will be either a predefined conversion from type $T$ to $T'$, or the predefined identity function defined over $T'$.

### 7.13.9 List expressions

A list expression of the form "`{V1, V2, ..., Vn}`" is syntactically replaced by "`insert (V1, insert (V2, ... insert (Vn, nil)...))`". For example, `{}` is converted to `nil` and `{1, 2, 3}` is converted to `insert (1, insert (2, insert (3, nil)))`.

It is worth noticing that this syntax notation is not restricted to list types. It can also be used for sorted lists, sets, and even any LNT type that has `nil` and `insert` operations with the right profiles. Precisely, the following constraints must be satisfied:

(VL1) All the elements V$i$ must be of the same type $T'$ and this type must be declared.

(VL2) The type $T$ of the list expression result must be declared.

(VL3) The `nil` function must be declared (with the profile specified in Section 5.7).

(VL4) The `insert` function must be declared (with the profile specified in Section 5.7).

To improve the runtime performance, if "`{V1, ..., Vn}`" has a sorted list or set type, then it is recommended to sort `V1`, ..., `Vn` such that if V$i$ `<` V$j$ holds and can be determined statically, then $i < j$.

# Chapter 8

# Process definitions in LNT

## 8.1 Notations

This chapter uses the BNF notations defined in Section 3.1 and the non-terminals defined in Chapters 5, 6, and 7.

The following additional conventions are used:

- $B$ is a behaviour
- $E$ is an event identifier (which represents either an input/output communication gate or an exception)
- $O$ is an offer
- $\Pi$ is a process identifier

## 8.2 Syntax

| | | | |
|---|---|---|---|
| *process_definition* | ::= | **process** $\Pi$ [ `[`$formal\_events_0$ `,` ... `,` $formal\_events_m$`]` ] <br> [ `(`$formal\_parameters_1$ `,` ... `,` $formal\_parameters_n$`)` ] **is** <br> $process\_pragma_1 \ldots process\_pragma_l$ <br> $precondition_1 \ldots precondition_j$ <br> $postcondition_1 \ldots postcondition_k$ <br> [$B$] <br> **end process** | *process definition* |

| | | | |
|---|---|---|---|
| *process_pragma* | ::= | `!virtual` | *virtual process* |
| | \| | `!implementedby "LOTOS:`*name*`"` | LOTOS *name* |

| | | | |
|---|---|---|---|
| $B$ | ::= | **null** | *no effect (with continuation)* |
| | \| | **stop** | *inaction (without continuation)* |
| | \| | $B_1$ `;` $B_2$ | *sequential composition* |
| | \| | $X$ `:=` $V$ | *deterministic assignment* |
| | \| | $X$ `:=` **any** $T$ [ **where** $V$ ] | *nondeterministic assignment* |
| | \| | [**eval**] $X$ `:=` $F$ [ `[`*actual_events*`]` ] `(`$actual\_parameter_1$`,`...`,` | *procedure call with result* |
| | \| | **eval** $F$ [ `[`*actual_events*`]` ] `(`$actual\_parameter_1$`,`...`,` | *procedure call without result* |
| | | $actual\_parameter_n$`)` | |
| | \| | $X$ `[`$V_0$`]` `:=` $V_1$ | *array element assignment* |
| | \| | **var** $var\_declaration_0$`,`...`,`$var\_declaration_n$ **in** | *variable declaration* |
| | | $B_0$ | |
| | | **end var** | |
| | \| | **case** $V_1$`,`...`,`$V_\ell$ | *case behaviour* |
| | | [ **var** $var\_declaration_0$`,`...`,`$var\_declaration_n$] **in** | |
| | | $match\_clause_0$ `->` $B_0$ | |
| | | `|` ... | |
| | | `|` $match\_clause_m$ `->` $B_m$ | |
| | | **end case** | |
| | \| | [**only**] **if** $V_0$ **then** $B_0$ | *conditional behaviour* |
| | | [ **elsif** $V_1$ **then** $B_1$ | |
| | | ... | |
| | | **elsif** $V_n$ **then** $B_n$ ] | |
| | | [ **else** $B_{n+1}$ ] | |
| | | **end if** | |
| | \| | **loop** | *forever loop* |
| | | $B_0$ | |
| | | **end loop** | |
| | \| | **loop** $L$ **in** | *breakable loop* |
| | | $B_0$ | |
| | | **end loop** | |
| | \| | **while** $V$ **loop** | *while loop* |
| | | $B_0$ | |
| | | **end loop** | |
| | \| | **while** $V$ **loop** $L$ **in** | *breakable while loop* |
| | | $B_0$ | |
| | | **end loop** | |
| | \| | **for** $B_0$ **while** $V$ **by** $B_1$ **loop** | *for loop* |
| | | $B_2$ | |
| | | **end loop** | |

| **for** $B_0$ **while** $V$ **by** $B_1$ **loop** $L$ **in** *breakable for loop*
  $B_2$
  **end loop**
| **break** $L$ *loop break*
| **use** $X_1$ , ... , $X_n$ *variable use*
| **access** $E_1$ , ... , $E_n$ *event access*
| **raise** $E$ [ ( $V_1$ , ... , $V_n$ ) ] *exception raise*
| **assert** $V$ [ **raise** $E$ [ ( $V_1$ , ... , $V_n$ ) ]] *assertion*
| $\Pi$ [ [*actual_events*] ] [ (*actual_parameter*$_1$ , ... , *actual_parameter*$_n$) ] *process call*
| $E$ [ ( $O_0$ , ... , $O_n$ ) ] [ **where** $V$ ] *communication*
| **alt** *nondeterministic choice*
  $B_0$
  [] ... []
  $B_n$
  **end alt**
| **par** [ $E_0$ , ... , $E_n$ **in** ] *parallel composition*
  [ $E_{(0,0)}$ , ... , $E_{(0,n_0)}$ -> ] $B_0$
  || ... ||
  [ $E_{(m,0)}$ , ... , $E_{(m,n_m)}$ -> ] $B_m$
  **end par**
| **hide** *event_declaration*$_0$ , ... , *event_declaration*$_n$ **in** *hiding*
  $B$
  **end hide**
| **disrupt** $B_1$ **by** $B_2$ **end disrupt** *disrupting*

$O$ ::= [ $X$ -> ] $V$ *output offer*
| [ $X$ -> ] ?$P$ *input offer*

## 8.3 Resolution of syntactic ambiguities

In a behaviour $B$, there can be a syntactic ambiguity between communications and process calls without event parameters. Here are examples of such ambiguous behaviours:

$Z$
$Z(1, 2)$
$Z(?X)$

This ambiguity is solved on the semantic level. If the identifier $Z$ is declared as a gate identifier in the current context, then the behaviour is considered to be a communication on gate $Z$. Otherwise, the behaviour is assumed to be a call to some process named $Z$.

Thus, priority is given to gate identifiers, meaning that, in process definitions, a formal gate parameter hides any process defined elsewhere with no event parameter.

## 8.4 Process definition

LNT (like LOTOS) allows a behaviour to be named using a *process definition*. A process is an object that denotes a behaviour; it can be parameterised by a list of formal events and a list of formal variables. Note that processes, like functions, cannot be parameters of processes: LNT is a first-order language.

A process definition consists of a process name, $\Pi$, optionally a list of formal event parameters $formal\_events_0, \ldots, formal\_events_m$, optionally a list of formal parameters $formal\_parameter_1, \ldots, formal\_parameter_n$, optionally a list of preconditions $precondition_1, \ldots, precondition_j$, optionally a list of postconditions $postcondition_1, \ldots, postcondition_k$, and a behaviour $B$ called the body of the process:

*process_definition* ::= **process** $\Pi$ [ `[`*formal_events*$_0$`,`...`,`*formal_events*$_m$`]` ]
[ `(`*formal_parameters*$_1$`,`...`,`*formal_parameters*$_n$`)` ] **is**
*process_pragma*$_1$...*process_pragma*$_l$
*precondition*$_1$...*precondition*$_j$
*postcondition*$_1$...*postcondition*$_k$
[$B$]
**end process**

A process can be defined without formal event parameters.

A process can be defined without formal value parameters.

The process names must be pairwise distinct. This means that (contrary to functions) overloading is not supported, even for processes with different parameter lists.

Each LNT process $\Pi$ is translated into a LOTOS process $\Pi$ of functionality "**exit** ($S_1$, ..., $S_n$)", where the $S_1$, ..., $S_n$ is the list of the sorts of the formal variable parameters of mode "**out**" or "**in out**".

The preconditions and postconditions of a process have the same meaning as in functions, the only difference being that the "**result**" keyword cannot occur in postconditions since there is no "**return**" behaviour. The description of preconditions and postconditions in Section 7.10 is still valid when replacing the word "function" by "process".

## 8.5 Process pragmas

The pragma "`!virtual`" expresses that the process is not fully defined in the current module but in another module that is not included (transitively) in the current module.

The pragma "`!implementedby "LOTOS:`$\Pi'$`"`" triggers the generation of a LOTOS process $\Pi'$ with the same parameters and functionality as the LOTOS process Π; the body of Π′ is always a call to the LOTOS process Π.

The following static semantics constraints apply to Π:

(PPG1) In the list *process_pragma*$_1$...*process_pragma*$_l$ of each process definition, there should be at most one pragma of each kind (i.e., there can neither be two "`!implementedby "LOTOS:..."`" pragmas not two "`!version`" pragmas).

(PPG2) The names provided by pragmas "`!implementedby "LOTOS:..."`" for all processes should be pairwise distinct.

(PPG3) To avoid name clashes in the generated LOTOS code, the name provided by a pragma "`!implementedby "LOTOS:..."`" should not be a name of another LNT process defined by the user. In particular, cyclic or self references such as "**process** `P` **is !implementedby** `"LOTOS:P"` ..." are forbidden. [checked by CÆSAR/CÆSAR.ADT]

(PPG4) If the body $B_0$ is empty then the pragma "`!virtual`" must be present.

(PPG5) If the pragma "`!virtual`" is present, the body $B_0$ should be either "**null**" or empty.

(PPG6) If the pragma "`!virtual`" is present, then $P$ must not have any other pragma.

(PPG7) A process $P$ may have at most one virtual definition (i.e., a definition with pragma "`!virtual`") and one actual definition (i.e., a definition without pragma "`!virtual`").

(PPG8) If a process $P$ has both a virtual and an actual definition, then those definitions should be located in distinct modules, and the module defining the actual process should not be included (transitively) in the module defining the virtual process.

(PPG9) If a process $P$ has both a virtual and an actual definition, then those definitions should have identical profiles, i.e., same event parameter names and channels, and same value parameter modes, names, and types. As regards parameter modes, the following exception holds: to an "**in var**" (resp. "**out var**") parameter in the actual definition corresponds an "**in**" (resp. "**out**") parameter in the virtual definition, i.e., the virtual definition should not contain "**in var**" and "**out var**" parameters.

In pragmas "`!implementedby "LOTOS:..."`", the prefix "`LOTOS:`" is case-sensitive. Other forms, such as "`Lotos:`" are rejected.

## 8.6 Lists of formal events

The rules governing formal event parameters for functions (see Section 7.7) also apply to formal event parameters for processes.

The key difference is that in processes, formal event parameters are not necessarily exceptions. Therefore, they may have a channel different from "**exit**" and they can even be untyped (i.e., be declared with the channel "**any**").

When an event occurs in a routine, its value is the actual event passed as argument to the routine. However, in the current version of LNT2LOTOS, this general rule has practical limitations when applied to events declared as exceptions.
Namely, if the root process is specified on the command line using the "`-root`" option, and if this process has formal event parameters declared as exceptions, then the value of each exception is the corresponding formal event parameter instead of the actual event passed on the command line, which is ignored.
For instance, if one uses option "**-root** "$P$ [$E_1$]"" and $P$ has formal event parameter $E_2$ declared as an exception, then the exception will be named $E_2$ instead of $E_1$.

## 8.7 Lists of formal parameters

The rules governing formal value parameters for functions (see Sections 7.8 and 7.9) apply also to formal value parameters for processes.

## 8.8 Behaviours

The control part and the data part of LNT are symmetrical: behaviours are extensions of statements except on the following points:

- The "**return**" statement has no behaviour counterpart.
- It is not mandatory that every unbreakable "**loop**" construct be eventually interrupted, since it is very common for a process to loop forever without exiting. Rule (UL1) is thus relaxed in the control part.

As a general principle, the rules given for statements in Section 7.11 also apply to behaviours having the same syntax as these statements. In order to avoid repetition, we only discuss here those behaviours that do not exist as statements or are slightly different.

### 8.8.1 Stop

The "**stop**" behaviour terminates the execution of the enclosing process.

Note: The termination performed by "**stop**" is said to be *unsuccessful*, as it is impossible for any other process to resume sequentially after "**stop**" (said differently, "**stop**" represents a deadlock). This is quite different from the *successful termination* performed by the "**null**" operator, since "**null**" offers a "$\delta$" action (see Section 6.6) that allows sequential continuation.

### 8.8.2 Procedure call

Procedure calls are considered similar to assignments: they execute instantaneously, and do not generate any transitions.

### 8.8.3 Only-if statement

Compared to the data part, the "**if**" construct is extended with the optional prefix "**only**" that is useful to implement guarded commands. Precisely, the behaviour

**only if** $V_0$ **then** $I_0$
**elsif** $V_1$ **then** $I_1$
...
**elsif** $V_n$ **then** $I_n$
**end if**

is syntactic sugar for

**if** $V_0$ **then** $I_0$
**elsif** $V_1$ **then** $I_1$
...
**elsif** $V_n$ **then** $I_n$
**else stop**
**end if**

(OIF1) An **only if** behaviour must not have an **else** branch.

Notice that a missing "**else**" branch in an "**if**" statement is equivalent to "**else null**" and, thus, non-blocking (i.e., if the conditions following "**if**" and "**elsif**" are all false, then the "**else**" can be executed). To the contrary, a missing "**else**" branch in an "**only if**" statement is equivalent to "**else stop**" and thus blocking. The "**only if**" statement is most useful as part of a "alt" statement.

### 8.8.4 Nondeterministic assignment

The behaviour "$X$ **:= any** $T$ [ **where** $V$ ]" assigns to variable $X$ an arbitrary value of type $T$ such that the value of expression $V$ is true; if variable $X$ occurs in $V$, it refers to the candidate value, and not to any prior value of $X$.

Note: Nondeterministic assignment can be used to express the choice of values, as in a LOTOS choice statement. For instance, the LOTOS behaviour "**choice** $X$ : $T$ `[]` $B$" can be written as "$X$ **:= any** $T$ **;** $B$".

The following static semantics constraints apply to this behaviour:

(NA1) The assignment should be "useful", i.e., there should exist at least one execution path on which the new value of $X$ is read before the execution completes and before $X$ is modified again (should it be).

### 8.8.5 Exception raise

The meaning of the "**raise**" construct in LNT processes is similar to that in LNT functions (see Section 7.11.4).

### 8.8.6 Assertion

The meaning of the "**assert**" construct in LNT processes is similar to that in LNT functions (see Section 7.11.5).

### 8.8.7 Process call

A process call has the form:

$\Pi$ [ `[`*actual_events*`]` ] [ `(`*actual_parameter*$_1$`,`...`,`*actual_parameter*$_n$`)` ]

Process calls have many analogies with procedure calls (see Sectionr̃efsec:procedure-call).

The actual events can be written either in the "positional" style or in the "named" one. In the named style:

- The notation "$E_{formal,i}$ `->` $E_{actual,i}$" means that the formal event parameter $E_{formal,i}$ of process $\Pi$ is instantiated with the actual event $E_{actual,i}$.
- The notation "`...`" means that each formal event parameter $E$ of $\Pi$ that does not appear in $E_{formal,1}$, ..., $E_{formal,n}$ is instantiated with the actual event $E$.

The static semantics constraints (AG1) to (AG2) apply to the positional style "$E_1$`,`...`,`$E_n$". The static semantics constraints (AG3) to (AG7) apply to the named style "$E_{formal,1}$ `->` $E_{actual,1}$`,`...`,`$E_{formal,n}$ `->` $E_{actual,n}$[`,...`]". The remaining constraints (AG8) to (AG9) apply to both positional and named styles.

(AG1) The number of actual event parameters of the process call must be equal to the number of formal event parameters of the corresponding process definition.

(AG2) Each actual event parameter $E_1$, ..., $E_n$ must have been declared in the current context (i.e., as a formal parameter of the process that contains the call to $\Pi$, or in an enclosing "**hide**" statement), except for the predefined exception "**unexpected**".

(AG3) The formal events $E_{formal,1}$, ..., $E_{formal,n}$ must be formal events of $\Pi$ and be pairwise distinct.

(AG4) Each actual event parameter $E_{actual,1}$, ..., $E_{actual,n}$ must have been declared in the current context (i.e., as a formal parameter of the process that contains the call to $\Pi$, or in an enclosing "**hide**" statement), except for the predefined exception "**unexpected**".

(AG5) If the notation "`...`" is used in "$E_{formal,1}$ `->` $E_{actual,1}$`,`...`,`$E_{formal,n}$ `->` $E_{actual,n}$`,...`", all the formal events of $\Pi$ that do not appear in $E_{formal,1}$, ..., $E_{formal,n}$ must correspond to formal event parameters of the process that contains the call to $\Pi$.

(AG6) When "`...`" is omitted, all the formal events of $\Pi$ must appear in $E_{formal,1}$, ..., $E_{formal,n}$.

(AG7) Process $\Pi$ must be defined in the current module, meaning that the named style can only be used to call processes defined in the same module (because, at present, LNT2LOTOS does not do sophisticated inter-module analysis). [checked by LNT2LOTOS]

(AG8) In either named or positional style, each actual event must be compatible (as defined in Section 6.7) with the corresponding formal event of the process definition.

(AG9) In either named or positional style, each actual event must be different from "**i**". [checked by LNT2LOTOS]

The following constraints apply to variable parameters:

(PI1) The constraints concerning actual parameters of procedure calls, i.e. (PC1), (PC2), (PC3), and (PC4), must be satisfied.

(PI2) A recursive process call must be *terminal*, i.e. must not be followed in sequence (meaning, according to sequential composition) by any further statement (except, possibly, the "**null**" statement). [checked by CÆSAR/CÆSAR.ADT]

(PI3) For each recursive process call, the list of actual variable parameters of mode "**out**" or "**in out**" of the called process must be equal to the list of the formal parameters of mode "**out**" (or "**out var**") or "**in out**" of the calling process.

(PI4) For each variable $X$ passed as actual variable parameter ("?$X$" or "!?$X$"), there should exist at least one execution path on which the new value assigned to $X$ is read before the execution completes and before $X$ is modified again (should it be).

Note: if constraints (PI2) and (PI3) are not respected, LNT2LOTOS might generate LOTOS code for recursive process calls that does not respect a restriction of the CÆSAR compiler, namely the absence of recursion on the left hand side of an enable operator ">>". To review all the possible cases, consider the call of a process $P$ in the body of a process $P_0$:

- If the call of $P$ is not terminal in $P_0$ and if $P$ may call $P_0$ recursively (either directly or transitively) — i.e., if constraint (PI2) is not satisfied —, CÆSAR will not accept the LOTOS code generated by LNT2LOTOS.
- If the call of $P$ is terminal in $P_0$:
  - If $P$ does not call $P_0$, LNT2LOTOS generates LOTOS code that is accepted by CÆSAR.
  - If $P$ may call $P_0$ recursively (either directly or transitively):
    - If constraint (PI3) is satisfied, LNT2LOTOS generates LOTOS code that is accepted by CÆSAR.
    - Otherwise, LNT2LOTOS generates LOTOS code that is rejected by CÆSAR.

The following examples illustrate these restrictions and show how to modify LNT source code to meet the above constraints (PI2) and (PI3):

- The recursive process:

  **process** *P* [*E*:**any**] **is**
    *E*; *P* [*E*];
    **stop**
  **end process**

  violates constraint (PI2) because the call of $P$ is followed by the non-**null** behaviour "**stop**"; $P$ can be written without the "**stop**" behaviour (which is never reached anyway) as:

  **process** *P* [*E*:**any**] **is**
    *E*; *P* [*E*]
  **end process**

Notice that replacing “**stop**” by “**null**” would also be correct:

**process** $P$ [$E$:**any**] **is**
  $E$; $P$ [$E$];
  **null**
**end process**

- The process:

**process** $P$ [$E$:**any**] (**out** $X$:Nat) **is**
  **var** $Y$:Nat **in**
    $E$ (?$X$); $P$ [$E$] (?$Y$)
  **end var**
**end process**

violates constraint (PI3) because the actual parameter (i.e., variable $Y$) given for the recursive call of $P$ is different from the formal parameter $X$; $P$ can be written without the unnecessary local variable `Y` as:

**process** $P$ [$E$:**any**] (**out** $X$:Nat) **is**
  $E$ (?$X$); $P$ [$E$] (?$X$)
**end process**

- The process:

**process** $P$ [$E$:**any**] (**out** $X$, $Y$: Nat) **is**
  $E$ (?$X$, ?$Y$); $P$ [$E$] (?$Y$, ?$X$)
**end process**

violates constraint (PI3) because the order of the actual parameters in the call of $P$ is not the same as the order of the formal parameters; $P$ can be written by explicitly inlining one call as:

**process** $P$ [$E$:**any**] (**out** $X$, $Y$: Nat) **is**
  $E$ (?$X$, ?$Y$); $E$ (?$Y$, ?$X$); $P$ [$E$] (?$Y$, ?$X$)
**end process**

- The mutually recursive processes $P$ and $Q$:

**process** $P$ [$E$:**any**] (**out var** $X$, $Y$:Nat) **is**
  $E$ (?$X$, ?$Y$);
  **if** ($X < Y$) **then**
    $Q$ [$E$] (?$X$, ?$Y$)
  **end if**
**end process**
**process** $Q$ [$E$:**any**] (**out** $X$, $Y$:Nat) **is**
  $P$ [$E$] (?$Y$, ?$X$)
**end process**

violate constraint (PI3) because the order of the actual parameters of the call of $P$ is different from the order of the formal parameters of $Q$; $Q$ can be inlined in the body of $P$:

```
process P [E:any] (out var X, Y:Nat) is
  E (?X, ?Y);
  if (X < Y) then
    E (?Y, ?X);
    if (Y < X) then
      P [E] (?X, ?Y)
    end if
  end if
end process
```

- The mutually recursive processes $P$ and $Q$:

```
process P [E:any] (out var X:Nat) is
  var Y:Nat in
    E (?X, ?Y);
    if (X < Y) then
      Q [E] (?X, ?Y)
    end if
  end var
end process
process Q [E:any] (out var X, Y:Nat) is
  Y := 2;
  P [E] (?X)
end process
```

violate constraint (PI3) because $P$ and $Q$ do not have the same number of parameters of mode "**out**"; $P$ and $Q$ can be rewritten by adding "dummy" variables:

```
process P [E:any] (out var X:Nat, out dummy:Nat) is
  var Y:Nat in
    E (?X, ?Y);
    if (X < Y) then
      Q [E] (?X, ?dummy)
    if (X < Y) then
      dummy := 0
    end if
  end var
end process
process Q [E:any] (out X, Y:Nat) is
  Y := 2;
  var dummy:Nat in
    P [E] (?X, ?dummy)
  end var
end process
```

The execution of a process call begins with the simultaneous evaluation of the expressions corresponding to the "**in**" parameters. For an "**in out**" parameter, the input value is the value of the variable given as the parameter. Then the body of the process is executed, substituting formal event parameters by actual event parameters. The body should assign all "**out**" parameters.

Note: each call of an LNT process is translated into an call of a LOTOS process of functionality "**exit** ($S_1$, ..., $S_n$)", where the $S_1$, ..., $S_n$ is the list of the sorts of the actual variable parameters of mode "**out**" or "**in out**".

### 8.8.8 Communication

In LNT, as in LOTOS, behaviours communicate by *rendezvous* on *gates*.

In LNT processes, gates are declared either as formal event parameters or using the "**hide**" operator.

The behaviour "$E$ [`(`$O_0$`,`...`,`$O_n$`)`] [**where** $V$]" waits for a rendezvous on gate $E$. The offers $O_0$, ..., $O_n$ describe the data exchanged during the rendezvous. An offer "$V$" corresponds to an emission (output) of value expression $V$. An offer "`?`$P$" corresponds to a reception (input) of a value matching pattern $P$; the variables of $P$ must be already declared. A rendezvous takes place only if the value expression in the condition "`[`$V$`]`" evaluates to true; condition $V$ can use values received by the offers $O_0$, ..., $O_n$.

The communication is blocked by both sending and receiving values: the behaviour waiting for a rendezvous is suspended and terminates immediately after the rendezvous takes place.

The internal gate "**i**" (see Section 6.6) specifies a non-observable action of the behaviour and terminates successfully.

In LNT, as in LOTOS, a rendezvous is symmetrical: there is no difference between the sender and the receiver. The rendezvous on a gate may allow several sending and receiving offers at the same time.

For gates that are not untyped, the list of offers must match one of the profiles of the channel with which $E$ was declared. In this case, the variable names "$X$" can be specified; if specified, they must be identical to the variable names of the channel profile.

(COM1) Variables used in all receptions "`?`$P$" of the same communication must be pairwise distinct across all receptions.

(COM2) The variables bound in the reception patterns "`?`$P$" should be used afterwards; see rule (PA4).

(COM3) The predefined gate "**i**" cannot be used with offers.

(COM4) If $E$ is typed, then its channel may not be the **exit** channel.

### 8.8.9 Nondeterministic choice (alternative)

The behaviour "**alt** $B_0$ `[]` ... `[]` $B_n$ **end alt**" (which was formerly written "**select** $B_0$ `[]` ... `[]` $B_n$ **end select**" until March 2024) may execute either $B_0$, or $B_1$, ..., or $B_n$ (where $n \geq 0$). The first action (e.g., rendezvous, internal action, or successful termination action) executed by any $B_i$ resolves the choice in favor of $B_i$.

### 8.8.10 Parallel composition

A parallel composition has the form:

**par** [ $E_0$`,`...`,`$E_n$ **in** ]

[ $E_{(0,0)}$ , ... , $E_{(0,n_0)}$ -> ] $B_0$
|| ... ||
[ $E_{(m,0)}$ , ... , $E_{(m,n_m)}$ -> ] $B_m$
**end par**

The set of events "$\{E_0, \ldots, E_n\}$" is called the *global synchronisation set*. Each event in this set must have been declared as a gate. If "$\{E_0, \ldots, E_n\}$" is omitted, then the global synchronisation set is empty. If a behaviour among $B_0, \ldots, B_m$ is waiting for a communication whose gate belongs to the global synchronisation set, then this communication can happen only if all behaviours $B_0, \ldots, B_m$ can make this communication simultaneously.

For all $i$ in $0..m$, the set of events "$\{E_{(i,0)}, \ldots, E_{(i,n_i)}\}$" is called the *local synchronisation set* of $B_i$. Each event in this set must have been declared as a gate. If "$E_{(i,0)}, \ldots, E_{(i,n_i)}$" is omitted, then the local synchronisation set of $B_i$ is empty. If a behaviour among $B_0, \ldots, B_m$ is waiting for a communication whose gate belongs to its local synchronisation set, then this communication can happen only if all behaviours $B_0, \ldots, B_m$ that contain this gate in their local synchronisation set can make this communication simultaneously.

If a behaviour among $B_0, \ldots, B_m$ is waiting for a communication whose gate does not belong to its local synchronisation set nor to the global synchronisation set, then this communication can happen without restriction.

(PAR1) Events that belong either to the global synchronisation set or to a local synchronisation set must be different from "**i**".

(PAR2) Events $E_0$, ..., $E_n$ must not appear in $E_{(0,0)}$, ..., $E_{(0,n_0)}$, ..., $E_{(m,0)}$, ..., $E_{(m,n_m)}$. [checked by LNT2LOTOS]

(PAR3) Every event that belongs to some $B_i$ but does not belong to the corresponding $\{E_{(i,0)}, ..., E_{(i,n_i)}\}$ must not belong to $\{E_{(0,0)}, ..., E_{(0,n_0)}, ..., E_{(m,0)}, ..., E_{(m,n_m)}\}$. [checked by LNT2LOTOS]

(PAR4) If a $B_i$ assigns a value to a variable or parameter, every $B_j$ such that $i \neq j$ must neither assign a value to that variable or parameter, nor read its value.

(PAR5) Behaviours $B_0, \ldots, B_n$ must not contain a recursive call (either direct or indirect) to the current process. [checked by CÆSAR]

(PAR6) Behaviours $B_0, \ldots, B_n$ must not contain a statement "**break** $L$" if the corresponding loop $L$ is not defined by one of the behaviours $B_0, \ldots, B_n$, *i.e.*, if the **par** statement is inside the body of the loop $L$. [checked by LNT2LOTOS]

(PAR7) Events that belong either to the global synchronisation set or to a local synchronisation set must be different from "**unexpected**".

(PAR8) Events that belong either to the global synchronisation set or to a local synchronisation set must have been declared as gates.

### 8.8.11 Hiding

**hide** *event_declaration*$_0$ , ... , *event_declaration*$_n$ **in**

$B$
**end hide**

The hiding operator declares a list of events $E_0, ..., E_n$, which are gates. Such gates are not observable from the environment of the behaviour: each communication (possibly with input/output offers) on a hidden gate $E_i$ is externally equivalent to the internal action "**i**".

(H1) The hidden events $E_0, ..., E_n$ must be pairwise distinct.

(H2) Each hidden event $E_i$ must be different from "**i**".

(H3) Each hidden event $E_i$ must be different from "**unexpected**".

(H4) The hidden events $E_0, ..., E_n$ may not be declared with the channel **exit**.

### 8.8.12 Disruption

**disrupt** $B_1$ **by** $B_2$ **end disrupt**

The **disrupt** behaviour starts behaviour "$B_1$", which executes normally. However, at any moment, "$B_1$" can be interrupted, in which case the execution of "$B_2$" starts and "$B_1$" is terminated. Yet, if "$B_1$" successfully terminates before any action has taken place in "$B_2$", the disrupt behaviour (as a whole) terminates, meaning that the possibility to be interrupted by "$B_2$" disappears.

(DIS1) Behaviour $B_1$ should not contain a recursive call (neither direct nor indirect) to the current process. [checked by CÆSAR]

# Appendix A

# Syntax summary of the LNT language (version 7.5)

## A.1 Extended BNF notation used in this appendix

| Notation | Meaning |
|---|---|
| [ $y$ ] | optional operator (0 or 1 instance of y) |
| $y_1 \| y_2$ | choice of either $y_1$ or $y_2$ |
| $y_0...y_n$ | concatenation of one or more $y$'s |
| $y_1...y_n$ | concatenation of zero or more $y$'s |
| $y_0, ..., y_n$ | concatenation of one or more $y$'s separated by commas |
| $y_1, ..., y_n$ | concatenation of zero or more $y$'s separated by commas |

## A.2 Identifiers

| Identifier (terminal symbol) | Meaning |
|---|---|
| $M$ | *module* |
| $T$ | *type* |
| $C$ | *type constructor* |
| $X$ | *variable* |
| $F$ | *function* |
| $L$ | *loop label* |
| $\Gamma$ | *channel* |
| $E$ | *event* |
| $\Pi$ | *process* |

NOTE: The following are not identifiers but are non-terminal symbols, and are defined below.

| Non-terminal symbol | Meaning |
|---|---|
| $I$ | *statement* |
| $V$ | *expression* |
| $P$ | *pattern* |
| $B$ | *behaviour* |
| $O$ | *offer* |

## A.3 Modules

*lnt_file* ::= **module** $M$[`(`$M_0, ..., M_m$`)`] *module*
  [**with** $predefined\_function_0, ..., predefined\_function_n$] **is**
  $module\_pragma_1 ... module\_pragma_p$
  $definition_0 ... definition_q$
**end module**

| | | | |
|---|---|---|---|
| *predefined_function* | ::= | `==` \| `=` | *equality* |
| | \| | `<>` \| `!=` | *inequality* |
| | \| | `<` | *less than* |
| | \| | `<=` | *less than or equal to* |
| | \| | `>` | *greater than* |
| | \| | `>=` | *greater than or equal to* |
| | \| | `append` | *tail insertion* |
| | \| | `card` | *set cardinality* |
| | \| | `delete` | *element deletion* |
| | \| | `diff` | *asymmetric difference* |
| | \| | `element` | *indexed access* |
| | \| | `empty` | *emptyness test* |
| | \| | `first` | *first element* |
| | \| | `get` | *field selection* |
| | \| | `head` | *first element* |
| | \| | `insert` | *insertion* |
| | \| | `inter` | *intersection* |
| | \| | `last` | *last element* |
| | \| | `length` | *list length* |
| | \| | `member` | *membership test* |
| | \| | `minus` | *symmetric difference* |
| | \| | `ord` | *ordinal* |
| | \| | `remove` | *element removal* |
| | \| | `reverse` | *reversal* |

| | | |
|---|---|---|
| | \| `set` | *field update* |
| | \| `subset` | *subset test* |
| | \| `tail` | *next elements* |
| | \| `union` | *union* |
| | \| `val` | *value* |

| | | | |
|---|---|---|---|
| *module_pragma* | ::= | `!nat_bits` *nat* | *number of bits for type Nat* |
| | \| | `!nat_inf` *nat* | *lowest value of type Nat* |
| | \| | `!nat_sup` *nat* | *highest value of type Nat* |
| | \| | `!nat_check` *bit* | *check for Nat overflows/underflows* |
| | \| | `!int_bits` *nat* | *number of bits for type Int* |
| | \| | `!int_inf` *int* | *lowest value of type Int* |
| | \| | `!int_sup` *int* | *highest value of type Int* |
| | \| | `!int_check` *bit* | *check for Int overflows/underflows* |
| | \| | `!num_bits` *nat* | *number of bits for numeral types* |
| | \| | `!num_card` *nat* | *maximal cardinality for numeral types* |
| | \| | `!string_card` *nat* | *maximal cardinality for type String* |
| | \| | `!update` *string* | *update tag* |
| | \| | `!version` *string* | *version tag* |

where *nat* denotes a natural number constant (in decimal notation without underscores), *int* denotes an integer number constant (in decimal notation without underscores), and *bit* denotes 0 or 1.

| | | | |
|---|---|---|---|
| *definition* | ::= | *type_definition* | *type definition* |
| | \| | *function_definition* | *function definition* |
| | \| | *channel_definition* | *channel definition* |
| | \| | *process_definition* | *process definition* |

## A.4 Types

| | | | |
|---|---|---|---|
| *type_definition* | ::= | **type** $T$ **is** *type_pragma*$_1$...*type_pragma*$_n$ | *type* |
| | | *type_expression* | |
| | | [**with** *predefined_function_declaration*$_0$, ..., *predefined_function_declaration*$_m$] | |
| | | **end type** | |

| | | | |
|---|---|---|---|
| *type_pragma* | ::= | `!external` | *external type* |
| | \| | `!implementedby "`[`C:`]*name*`"` | *C type name* |
| | \| | `!comparedby "`[`C:`]*name*`"` | *C equality function* |
| | \| | `!printedby "`[`C:`]*name*`"` | *C printing function* |

| | | |
|---|---|---|
| \| | `!list` | *print as list* |
| \| | `!iteratedby "`[`C:`]$name_1$`" , "`[`C:`]$name_2$`"` | *C iterator functions* |
| \| | `!pointer` | *C pointer implementation* |
| \| | `!nopointer` | *C unboxed implementation* |
| \| | `!bits` *nat* | *number of bits for the C type* |
| \| | `!card` *nat* | *maximal cardinality for the C type* |

*string* ::= *"character*"*

| | | | |
|---|---|---|---|
| *type_expression* | ::= | $constructor\_definition_0, ..., constructor\_definition_n$ | *constructed type* |
| | \| | **set of** $T$ | *set* |
| | \| | **list of** $T$ | *list* |
| | \| | **sorted list of** $T$ | *sorted list* |
| | \| | **array** `[`$m$`..`$n$`]` **of** $T$ | *array* |
| | \| | **range** $m$`..`$n$ **of** $T'$ | *range* |
| | \| | $X:T'$ [**where** $V$] | *predicate* |
| | \| | | *empty (external type only)* |

*constructor_definition* ::= $C$ [`(`$constructor\_parameters_1, ..., constructor\_parameters_n$`)`]
$constructor\_pragma_1 ... constructor\_pragma_m$

*constructor_parameters* ::= $X_0, ..., X_n : T$ — *constructor parameters*

*constructor_pragma* ::= `!implementedby "`[`C:`]*name*`"` — *C operator name*

*predefined_function_declaration* ::= *predefined_function*
[$predefined\_function\_pragma_1 ... predefined\_function\_pragma_n$]

*predefined_function_pragma* ::= `!external` — *external function*
`!implementedby "`[(`C` | `LOTOS`)`:`]*name*`"` — *C/LOTOS name scheme*

## A.5 Channels

*channel_definition* ::= **channel** $\Gamma$ **is** [**raise**] — *channel definition*
$channel\_profile_0$,
...,
$channel\_profile_n$

**end channel**

*channel_profile* ::= `(`*profile_parameters*$_1$ `,` ... `,` *profile_parameters*$_n$`)` *channel profile*

*profile_parameters* ::= $X_0$ `,` ... `,` $X_n$ `:` $T$ *profile parameter list*

## A.6 Functions

*function_definition* ::= **function** $F$ [ `[`*formal_events*$_0$ `,` ... `,` *formal_events*$_m$`]` ]
[ `(`*formal_parameters*$_1$ `,` ... `,` *formal_parameters*$_n$`)` ] [ `:`$T$ ] **is**
*function_pragma*$_1$...*function_pragma*$_l$
*precondition*$_1$...*precondition*$_j$
*postcondition*$_1$...*postcondition*$_k$
[ $I_0$ ]
**end function** *function definition*

*formal_events* ::= *event_declaration* *formal events*

*event_declaration* ::= $E_0$ `,` ... `,` $E_n$ `:` $\Gamma$ *typed event declaration*
| $E_0$ `,` ... `,` $E_n$ `:` **any** *untyped event declaration*

*formal_parameters* ::= *parameter_mode* $X_0$ `,` ... `,` $X_n$`:`$T$ *formal parameters*

*parameter_mode* ::= [ **in** ] *input formal parameter*
| **in var** *input formal parameter used as local variable*
| **out** *output formal parameter*
| **out var** *output formal parameter used as local variable*
| **in out** *input / output formal parameter*

*precondition* ::= **require** $V$ [ **raise** $E$ [ `()` ] ]`;` *precondition*

*postcondition* ::= **ensure** $V$ [ **raise** $E$ [ `()` ] ]`;` *postcondition*

*function_pragma* ::= `!external` *external function*
| `!implementedby "`[(`C` | `LOTOS`)`:`]*name*`"` *C or* LOTOS *name scheme*

## A.7 Instructions and statements

$I$ ::= **null** — *no effect*
| $I_1$ ; $I_2$ — *sequential composition*
| **return** [ $V$ ] — *return*
| **raise** $E$ [ () ] — *exception raise*
| **assert** $V$ [ **raise** $E$ [ () ] ] — *assertion*
| $X$ := $V$ — *assignment*
| $X$ [$V_0$] := $V_1$ — *array element assignment*
| [ **eval** ] [ $X$ := ] $F$ [ [*actual_events*] ] (*actual_parameter*$_1$, ..., *actual_parameter*$_n$) — *procedure call*
| **var** *var_declaration*$_0$, ..., *var_declaration*$_n$ **in**
    $I_0$
  **end var** — *variable declaration*
| **case** $V_0$, ..., $V_\ell$ — *case statement*
    [ **var** *var_declaration*$_0$, ..., *var_declaration*$_n$] **in**
      *match_clause*$_0$ -> $I_0$
     | ...
     | *match_clause*$_m$ -> $I_m$
  **end case**
| **if** $V_0$ **then** $I_0$ — *conditional statement*
  [ **elsif** $V_1$ **then** $I_1$
    ...
    **elsif** $V_n$ **then** $I_n$ ]
  [ **else** $I_{n+1}$ ]
  **end if**
| **loop** — *forever loop*
    $I_0$
  **end loop**
| **loop** $L$ **in** — *breakable loop*
    $I_0$
  **end loop**
| **while** $V$ **loop** — *while loop*
    $I_0$
  **end loop**
| **while** $V$ **loop** $L$ **in** — *breakable while loop*
    $I_0$
  **end loop**
| **for** $I_0$ **while** $V$ **by** $I_1$ **loop** — *for loop*
    $I_2$

| | | | |
|---|---|---|---|
| | | **end loop** | |
| | $\mid$ | **for** $I_0$ **while** $V$ **by** $I_1$ **loop** $L$ **in** | *breakable for loop* |
| | | $I_2$ | |
| | | **end loop** | |
| | $\mid$ | **break** $L$ | *loop break* |
| | $\mid$ | **use** $X_0, \ldots, X_n$ | *variable use* |
| | $\mid$ | **access** $E_0, \ldots, E_n$ | *event access* |
| *var_declaration* | ::= | $X_0, \ldots, X_n : T$ | *variable list* |
| *actual_events* | ::= | $E_1, \ldots, E_n$ | *positional style* |
| | $\mid$ | $E_{formal,1}$ `->` $E_{actual,1}, \ldots, E_{formal,n}$ `->` $E_{actual,n}$ [`,...`] | *named style* |
| *actual_parameter* | ::= | $V$ | *actual parameter "in"* |
| | $\mid$ | `?`$X$ | *actual parameter "out"* |
| | $\mid$ | `!?`$X$ | *actual parameter "in out"* |
| *match_clause* | ::= | $P_0, \ldots, P_\ell$ [ **where** $V_0$ ] `|` ... `|` $P_n$ [ **where** $V_n$ ] | *match clause* |
| | $\mid$ | **any**, ..., **any** [ **where** $V$ ] | *wildcard* |

## A.8 Patterns

| | | | |
|---|---|---|---|
| $P$ | ::= | $X$ | *variable* |
| | $\mid$ | **any** $T$ | *wildcard* |
| | $\mid$ | $X$ **as** $P_0$ | *aliasing* |
| | $\mid$ | $C$ [ `(`$P_0, \ldots, P_n$`)` ] | *constructed pattern* |
| | $\mid$ | $P_1$ $C$ $P_2$ | *infix constructed pattern* |
| | $\mid$ | $F$ [ `(`$P_0, \ldots, P_n$`)` ] | *constant pattern* |
| | $\mid$ | $P_1$ $F$ $P_2$ | *infix constant pattern* |
| | $\mid$ | $P_0$ **of** $T$ | *type coercion* |
| | $\mid$ | `(`$P$`)` | *parenthesized pattern* |
| | $\mid$ | `{`$P_1, \ldots, P_n$`}` | *list pattern* |

## A.9 Value expressions

| | | | |
|---|---|---|---|
| $V$ | ::= | $X$ | *variable* |

| | | |
|---|---|---|
| \| | $X$ **.in** | *input parameter value (in postcondition only)* |
| \| | $X$ **.out** | *output parameter value (in postcondition only)* |
| \| | **result** | *function result (in postcondition only)* |
| \| | $C$ [ `(`$V_1$`,`...`,`$V_n$`)` ] | *constructor call* |
| \| | $V_1$ $C$ $V_2$ | *infix constructor call* |
| \| | $F$ [ `[`*actual_events*`]` ] [ `(`$V_1$`,`...`,`$V_n$`)` ] | *function call* |
| \| | $V_1$ $F$ [ `[`*actual_events*`]` ] $V_2$ | *infix function call* |
| \| | $V$ `.`[ `[`$E$`]` ] *field* | *field selection* |
| \| | $V$ `.`[ `[`$E$`]` ] `{`*field*$_0$`->`$V_0$`,`...`,`*field*$_n$`->`$V_n$`}` | *field update* |
| \| | $V_0$ `[` $V_1$ `]` | *array element access* |
| \| | $V$ **of** $T$ | *type coercion* |
| \| | `(`$V$`)` | *parenthesized expression* |
| \| | `{`$V_1$`,`...`,`$V_n$`}` | *list expression* |

## A.10 Processes

*process_definition* ::= **process** $\Pi$ [ `[`*formal_events*$_0$`,`...`,`*formal_events*$_m$`]` ]
[ `(`*formal_parameters*$_1$`,`...`,`*formal_parameters*$_n$`)` ] **is**
*process_pragma*$_1$...*process_pragma*$_l$
*precondition*$_1$...*precondition*$_j$
*postcondition*$_1$...*postcondition*$_k$
[$B$]
**end process** — *process definition*

| | | | |
|---|---|---|---|
| *process_pragma* | ::= | `!virtual` | *virtual process* |
| | \| | `!implementedby "LOTOS:`*name*`"` | LOTOS *name* |

## A.11 Behaviours

| | | | |
|---|---|---|---|
| $B$ | ::= | **null** | *no effect (with continuation)* |
| | \| | **stop** | *inaction (without continuation)* |
| | \| | $B_1$ `;` $B_2$ | *sequential composition* |
| | \| | $X$ `:=` $V$ | *deterministic assignment* |
| | \| | $X$ `:=` **any** $T$ [ **where** $V$ ] | *nondeterministic assignment* |
| | \| | [**eval**] $X$ `:=` $F$ [ `[`*actual_events*`]` ] `(`*actual_parameter*$_1$`,`...`,` | *procedure call with result* |
| | \| | **eval** $F$ [ `[`*actual_events*`]` ] `(`*actual_parameter*$_1$`,`...`,` | *procedure call without result* |

$actual\_parameter_n$)
| $X$ [$V_0$] := $V_1$ *array element assignment*
| **var** $var\_declaration_0$ , ... , $var\_declaration_n$ **in** *variable declaration*
$B_0$
**end var**
| **case** $V_1$ , ... , $V_\ell$ *case behaviour*
[ **var** $var\_declaration_0$ , ... , $var\_declaration_n$] **in**
$match\_clause_0$ -> $B_0$
| ...
| $match\_clause_m$ -> $B_m$
**end case**
| [**only**] **if** $V_0$ **then** $B_0$ *conditional behaviour*
[ **elsif** $V_1$ **then** $B_1$
...
**elsif** $V_n$ **then** $B_n$ ]
[ **else** $B_{n+1}$ ]
**end if**
| **loop** *forever loop*
$B_0$
**end loop**
| **loop** $L$ **in** *breakable loop*
$B_0$
**end loop**
| **while** $V$ **loop** *while loop*
$B_0$
**end loop**
| **while** $V$ **loop** $L$ **in** *breakable while loop*
$B_0$
**end loop**
| **for** $B_0$ **while** $V$ **by** $B_1$ **loop** *for loop*
$B_2$
**end loop**
| **for** $B_0$ **while** $V$ **by** $B_1$ **loop** $L$ **in** *breakable for loop*
$B_2$
**end loop**
| **break** $L$ *loop break*
| **use** $X_1$ , ... , $X_n$ *variable use*
| **access** $E_1$ , ... , $E_n$ *event access*
| **raise** $E$ [ ($V_1$ , ... , $V_n$) ] *exception raise*
| **assert** $V$ [ **raise** $E$ [ ($V_1$ , ... , $V_n$) ]] *assertion*

| | | | |
|---|---|---|---|
| | $\mid$ | $\Pi$ [ `[`*actual_events*`]` ] [ `(`$actual\_parameter_1$ `,`...`,` $actual\_parameter_n$`)` ] | *process call* |
| | $\mid$ | $E$ [ `(`$O_0$`,`...`,`$O_n$`)` ] [ **where** $V$ ] | *communication* |
| | $\mid$ | **alt**<br>  $B_0$<br>  `[]` ... `[]`<br>  $B_n$<br>**end alt** | *nondeterministic choice* |
| | $\mid$ | **par** [ $E_0$`,`...`,`$E_n$ **in** ]<br>  [ $E_{(0,0)}$`,`...`,`$E_{(0,n_0)}$ `->` ] $B_0$<br>  $\|$ ... $\|$<br>  [ $E_{(m,0)}$`,`...`,`$E_{(m,n_m)}$ `->` ] $B_m$<br>**end par** | *parallel composition* |
| | $\mid$ | **hide** $event\_declaration_0$`,`...`,` $event\_declaration_n$ **in**<br>  $B$<br>**end hide** | *hiding* |
| | $\mid$ | **disrupt** $B_1$ **by** $B_2$ **end disrupt** | *disrupting* |

| | | | |
|---|---|---|---|
| $O$ | ::= | [ $X$ `->` ] $V$ | *output offer* |
| | $\mid$ | [ $X$ `->` ] `?`$P$ | *input offer* |

# Appendix B

# Formal semantics of the LNT language (version 7.5)

## B.1 Preliminaries

We define the dynamic semantics of LNT programs, using formal SOS (*Structural Operational Semantics*) rules. Programs are assumed to have successfully passed all static analysis phases, such as *parsing*, *syntactic sugar expansion*, *binding analysis*, *typing analysis*, and *variable initialisation analysis*, thus enabling a simplified abstract syntax, which is also precisely defined in this annex.

### B.1.1 SOS rules

We give here a (partial) definition of SOS rules to fix the notations. The general goal of a set of SOS rules is to define an $n$-ary relation $R(e_1, \ldots, e_n)$ between elements $e_i$ $(i \in 1..n)$ of different sorts. In the sequel, the term *Boolean statement* denotes either a Boolean predicate in first-order logic, or an expression of the form $R\,(e_1, \ldots, e_n)$. Each SOS rule has the following form:

$$\frac{\mathit{Premise}_1 \ \ldots \ \mathit{Premise}_m}{\mathit{Conclusion}}$$

for some $m \geq 0$. The upper part "$\mathit{Premise}_1 \ \ldots \ \mathit{Premise}_m$" denotes a set of $m$ Boolean statements, and the lower part "*Conclusion*" denotes a single Boolean statement of the form $R\,(e_1, \ldots, e_n)$. The meaning is that the conclusion $R\,(e_1, \ldots, e_n)$ holds if each $\mathit{Premise}_i$ (for all i ranging in the interval $1..m$) itself holds, either logically if $\mathit{Premise}_i$ is a Boolean predicate, or by repetitive application of the SOS rules otherwise. All variables which occur free in some $\mathit{Premise}_i$ and/or in *Conclusion* are (implicitly) quantified universally over the whole rule.

In this appendix, we may use the concise notation "$\mathit{Premise}\,[i]$ $(i \in 1..m)$", where $\mathit{Premise}\,[i]$ is any Boolean statement that may depend on $i$, as a shorthand notation for the developed set of premises "$\mathit{Premise}\,[1] \ \ldots \ \mathit{Premise}\,[m]$", where each $\mathit{Premise}\,[k]$ (for any $k \in 1..m$) denotes $\mathit{Premise}\,[i]$ in which $i$ is replaced by $k$.

We may also use the following notation for sets of SOS rules:

$$\frac{\mathit{Premise}_1\,[j]\ \ldots\ \mathit{Premise}_n\,[j]}{\mathit{Conclusion}\,[j]}\quad (j \in 1..p)$$

where both $\mathit{Conclusion}\,[j]$ and each $\mathit{Premise}_i\,[j]$ $(i \in 1..m)$ are Boolean statements that may depend on $j$. This notation is equivalent to the set of $p$ rules obtained by replacing $j$ by numbers in the interval $1..p$, namely:

$$\frac{\mathit{Premise}_1\,[1]\ \ldots\ \mathit{Premise}_n\,[1]}{\mathit{Conclusion}\,[1]}$$

...

$$\frac{\mathit{Premise}_1\,[p]\ \ldots\ \mathit{Premise}_n\,[p]}{\mathit{Conclusion}\,[p]}$$

### B.1.2 Values and stores

The following notions of *value* and *store* are used in the SOS rules:

- A *value* is a ground term (i.e., a term without variables) containing only constructors. We write $\mathcal{V}$ for the set of all values and $v, v_0, v_1, \ldots$ for individual values.
- A *store* is a partial function from variables to values. We write $\sigma, \sigma_0, \sigma_1, \ldots$ for stores. The notation "$[X_1 \leftarrow v_1, \ldots, X_n \leftarrow v_n]$" (where $n \geq 0$, and $i \neq j \Longrightarrow X_i \neq X_j$) represents the store $\sigma$ such that $\sigma(X_1) = v_1, \ldots, \sigma(X_n) = v_n$ and $\sigma(X)$ is undefined for any $X \notin X_1, \ldots, X_n$. In particular, "$[]$" represents the empty store.
- Given two stores $\sigma_1$ and $\sigma_2$, we write "$\sigma_1 \oslash \sigma_2$" for the *update* of $\sigma_1$ with respect to $\sigma_2$, which consists of $\sigma_2$ plus the part of $\sigma_1$ corresponding to variables not overwritten by $\sigma_2$. Store update is formally defined as follows:

$$(\sigma_1 \oslash \sigma_2)(X) = \begin{cases} \sigma_2(X) & \text{if } \sigma_2(X) \text{ is defined} \\ \sigma_1(X) & \text{if } \sigma_2(X) \text{ is not defined and } \sigma_1(X) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

- Given two stores $\sigma_1$ and $\sigma_2$, we write "$\sigma_1 \oplus \sigma_2$" for the *disjoint union* of $\sigma_1$ and $\sigma_2$. Formally, $\sigma_1 \oplus \sigma_2$ is defined as $\sigma_1 \oslash \sigma_2$ only if the sets of variables defined in $\sigma_1$ and $\sigma_2$ are disjoint, and it is undefined otherwise.
- Given two stores $\sigma_1$ and $\sigma_2$, we write "$\sigma_1 \ominus \sigma_2$" for the *difference* between $\sigma_1$ and $\sigma_2$, which consists of the part of $\sigma_1$ corresponding to variables that are either not defined or defined with a different value in $\sigma_2$. Store difference is formally defined as follows:

$$(\sigma_1 \ominus \sigma_2)(X) = \begin{cases} \sigma_1(X) & \text{if } \sigma_1(X) \text{ is defined and either } \sigma_2(X) \text{ is not defined or } \sigma_2(X) \neq \sigma_1(X) \\ \text{undefined} & \text{otherwise} \end{cases}$$

## B.2 Dynamic semantics of expressions

### B.2.1 Definitions

The dynamic semantics of expressions are defined as a relation of the form "$\langle V, \sigma \rangle \rightarrow_{\mathsf{e}} v$", where $V$ is an expression, $\sigma$ is a store, and $v$ is a value. This relation means that in store $\sigma$, the expression $V$ evaluates to the value $v$. We assume the following:

- After parsing, parenthesized expressions have been eliminated.
- After syntactic sugar elimination, infix function (respectively, constructor) calls have been replaced by prefix function (respectively, constructor) calls; field selections, field updates, and array selections have been replaced by built-in functions (whose semantics are standard and not defined explicitly here); and list expressions have been replaced by appropriate constructor calls.
- After typing analysis, type coercions have been removed.
- After binding analysis, named parameter passing has been replaced by positional parameter passing.

We thus consider the following abstract syntax of expressions:

$$
\begin{array}{lcl}
V & ::= & X \\
 & \mid & C \; \texttt{(} V_1 \texttt{,} \; \ldots \texttt{,} \; V_n \texttt{)} \\
 & \mid & F \; \texttt{(} V_1 \texttt{,} \; \ldots \texttt{,} \; V_n \texttt{)}
\end{array}
$$

### B.2.2 Variable

The value of a variable $X$ is that recorded in the current store.

$$\frac{}{\langle X, \sigma \rangle \rightarrow_{\mathsf{e}} \sigma(X)}$$

### B.2.3 Constructor call

The value of "$C \; \texttt{(} V_1 \texttt{,} \; \ldots \texttt{,} \; V_n \texttt{)}$" is $C$ applied to the values of $V_1, \ldots, V_n$.

$$\frac{\langle V_i, \sigma \rangle \rightarrow_{\mathsf{e}} v_i \;\; (i \in 1..n)}{\langle C \; \texttt{(} V_1 \texttt{,} \; \ldots \texttt{,} \; V_n \texttt{)}, \sigma \rangle \rightarrow_{\mathsf{e}} C \; (v_1, \ldots, v_n)}$$

### B.2.4 Built-in function call

The value of "$F$ `(`$V_1$`, ... ,` $V_n$`)`" is $F$ applied to the values of $V_1, \ldots, V_n$. Here, $F$ is understood as a mathematical function, i.e., $F$ applied to values is itself a value defined mathematically.

$$\frac{\langle V_i, \sigma\rangle \rightarrow_{\mathsf{e}} v_i \ (i \in 1..n)}{\langle F \ \texttt{(}V_1\texttt{, ... ,}\ V_n\texttt{)}, \sigma\rangle \rightarrow_{\mathsf{e}} F\ (v_1, \ldots, v_n)}$$

### B.2.5 User-defined function call

When $F$ is used in an expression, the static semantics ensure that it contains neither "**out**" nor "**in out**" formal parameters. We thus assume that it is defined (omitting formal parameter types) as follows:

**function** $F$ `(`**in** $X_1$`, ... ,` $X_m$`)` `:` $T$ **is** $I$ **end function**

The value of "$F\ (V_1, \ldots, V_m)$" is the value returned after executing the body $I$ of $F$ in a store associating the value of $V_i$ to each formal parameter $X_i$ ($i \in 1..n$). Note that the SOS rule below anticipates on the dynamic semantics of statements (relation $\longrightarrow_{\mathsf{s}}$), defined in Section B.5.

$$\frac{\langle V_i, \sigma\rangle \rightarrow_{\mathsf{e}} v_i \ (i \in 1..n) \quad \langle I, [X_1 \leftarrow v_1, \ldots, X_m \leftarrow v_m]\rangle \xrightarrow{\mathbf{ret}(v)}_{\mathsf{s}} \sigma'}{\langle F\ (V_1, \ldots, V_m), \sigma\rangle \rightarrow_{\mathsf{e}} v}$$

## B.3 Dynamic semantics of patterns

### B.3.1 Definitions

Given a pattern $P$, a value $v$, and a store $\sigma$, the dynamic semantics of patterns are defined as a relation that has two possible forms:

- "$\langle P \sharp\ v, \sigma\rangle \rightarrow_{\mathsf{p}} \sigma'$", where $\sigma'$ is a store, means that in store $\sigma$, the pattern $P$ matches the value $v$, producing the updated store $\sigma'$.
- "$\langle P \sharp\ v, \sigma\rangle \rightarrow_{\mathsf{p}}$ **fail**" means that in store $\sigma$, the pattern $P$ does not match the value $v$.

We assume the following:

- After parsing, parenthesized patterns have been eliminated.
- After syntactic sugar elimination, infix constant (respectively constructor) patterns have been replaced by prefix constant (respectively constructor) patterns, and list patterns have been replaced by appropriate constructed patterns.
- After typing analysis, type coercions have been removed.

- After binding analysis, named parameter passing has been replaced by positional parameter passing.

We thus consider the following abstract syntax of patterns, which also merges the definition of *match_clause*:

$$
\begin{array}{lcl}
P & ::= & X \\
 & | & \textbf{any} \\
 & | & X \textbf{ as } P_0 \\
 & | & C \; \texttt{(} P_1 \texttt{,} \; \ldots \texttt{,} \; P_n \texttt{)} \\
 & | & F \; \texttt{(} V_1 \texttt{,} \; \ldots \texttt{,} \; V_n \texttt{)} \\
 & | & P_0 \textbf{ where } V \\
 & | & P_1 \; \texttt{|} \; P_2
\end{array}
$$

### B.3.2 Variable

A variable $X$ always matches any value $v$, which becomes the new value of $X$.

$$\frac{}{\langle X \sharp v, \sigma \rangle \rightarrow_{\mathsf{p}} \sigma \oslash [X \leftarrow v]}$$

### B.3.3 Wildcard

The wildcard **any** always matches any value $v$, the store being left unchanged.

$$\frac{}{\langle \textbf{any} \sharp v, \sigma \rangle \rightarrow_{\mathsf{p}} \sigma}$$

### B.3.4 Aliasing

A pattern “$X$ **as** $P_0$” matches a value $v$ if and only if $P_0$ matches $v$. In this case, $v$ becomes the new value of $X$.

$$\frac{\langle P_0 \sharp v, \sigma \rangle \rightarrow_{\mathsf{p}} \sigma'}{\langle X \textbf{ as } P_0 \sharp v, \sigma \rangle \rightarrow_{\mathsf{p}} \sigma' \oslash [X \leftarrow v]}$$

$$\frac{\langle P_0 \sharp v, \sigma \rangle \rightarrow_{\mathsf{p}} \textbf{fail}}{\langle X \textbf{ as } P_0 \sharp v, \sigma \rangle \rightarrow_{\mathsf{p}} \textbf{fail}}$$

### B.3.5 Constructed pattern

A pattern "$C$ ($P_1$, ..., $P_n$)" matches a value $v$ if and only if $v$ has the form "$C\ (v_1, \ldots, v_n)$" and every pattern $P_i$ matches the corresponding value $v_i$ ($i \in 1..n$).

$$\frac{\sigma_0 = \sigma \qquad \langle P_i \sharp v_i, \sigma_{i-1}\rangle \to_{\mathsf{p}} \sigma_i \ (i \in 1..n)}{\langle C\ \texttt{(}P_1\texttt{,}\ \ldots\texttt{,}\ P_n\texttt{)} \sharp C\ (v_1, \ldots, v_n), \sigma\rangle \to_{\mathsf{p}} \sigma_n}$$

$$\frac{\sigma_0 = \sigma \quad \langle P_i \sharp v_i, \sigma_{i-1}\rangle \to_{\mathsf{p}} \sigma_i \ (i \in 1..j-1) \quad \langle P_j \sharp v_j, \sigma_{j-1}\rangle \to_{\mathsf{p}} \textbf{fail}}{\langle C\ \texttt{(}P_1\texttt{,}\ \ldots\texttt{,}\ P_n\texttt{)} \sharp C\ (v_1, \ldots, v_n), \sigma\rangle \to_{\mathsf{p}} \textbf{fail}} \quad (j \in 1..n)$$

The above rules ensure that patterns are evaluated from left to right. Theoretically, this would allow a variable bound in a pattern to be used in an expression (e.g., a Boolean condition) located further to the right, although for practical reasons, this is not currently allowed by the static semantics.

$$\frac{(C \neq C') \vee (n \neq m)}{\langle C\ \texttt{(}P_1\texttt{,}\ \ldots\texttt{,}\ P_n\texttt{)} \sharp C'\ (v_1, \ldots, v_m), \sigma\rangle \to_{\mathsf{p}} \textbf{fail}}$$

### B.3.6 Constant pattern

A constant pattern of the form "$F$ ($V_1$, ..., $V_n$)" matches a value $v$ if and only if the expression "$F$ ($V_1$, ..., $V_n$)" (which contains no variable and can thus be evaluated in the empty store) evaluates to $v$. The store is left unchanged.

$$\frac{\langle F\ \texttt{(}V_1\texttt{,}\ \ldots\texttt{,}\ V_n\texttt{)}, []\rangle \to_{\mathsf{e}} v}{\langle F\ \texttt{(}V_1\texttt{,}\ \ldots\texttt{,}\ V_n\texttt{)} \sharp v, \sigma\rangle \to_{\mathsf{p}} \sigma}$$

$$\frac{\langle F\ \texttt{(}V_1\texttt{,}\ \ldots\texttt{,}\ V_n\texttt{)}, []\rangle \to_{\mathsf{e}} v' \quad v' \neq v}{\langle F\ \texttt{(}V_1\texttt{,}\ \ldots\texttt{,}\ V_n\texttt{)} \sharp v, \sigma\rangle \to_{\mathsf{p}} \textbf{fail}}$$

### B.3.7 Conditional pattern

A conditional pattern "$P_0$ **where** $V$" matches a value $v$ if and only if $P_0$ matches $v$ and $V$ evaluates to **true** in the resulting store.

$$\frac{\langle P_0 \sharp v, \sigma\rangle \to_{\mathsf{p}} \sigma' \quad \langle V, \sigma'\rangle \to_{\mathsf{e}} \mathbf{true}}{\langle P_0 \ \mathbf{where}\ V \sharp v, \sigma\rangle \to_{\mathsf{p}} \sigma'}$$

$$\frac{\langle P_0 \sharp v, \sigma\rangle \to_{\mathsf{p}} \sigma' \quad \langle V, \sigma'\rangle \to_{\mathsf{e}} \mathbf{false}}{\langle P_0 \ \mathbf{where}\ V \sharp v, \sigma\rangle \to_{\mathsf{p}} \mathbf{fail}}$$

$$\frac{\langle P_0 \sharp v, \sigma\rangle \to_{\mathsf{p}} \mathbf{fail}}{\langle P_0 \ \mathbf{where}\ V \sharp v, \sigma\rangle \to_{\mathsf{p}} \mathbf{fail}}$$

### B.3.8 Alternative

An alternative "$P_1 \mid P_2$" matches a value $v$ if and only if $P_1$ matches $v$ or else $P_2$ matches $v$. The patterns are evaluated from left to right, so that the resulting store is defined non-ambiguously if both patterns match $v$.

$$\frac{\langle P_1 \sharp v, \sigma\rangle \to_{\mathsf{p}} \sigma_1}{\langle P_1 \mid P_2 \sharp v, \sigma\rangle \to_{\mathsf{p}} \sigma_1}$$

$$\frac{\langle P_1 \sharp v, \sigma\rangle \to_{\mathsf{p}} \mathbf{fail} \quad \langle P_2 \sharp v, \sigma\rangle \to_{\mathsf{p}} \sigma_2}{\langle P_1 \mid P_2 \sharp v, \sigma\rangle \to_{\mathsf{p}} \sigma_2}$$

$$\frac{\langle P_1 \sharp v, \sigma\rangle \to_{\mathsf{p}} \mathbf{fail} \quad \langle P_2 \sharp v, \sigma\rangle \to_{\mathsf{p}} \mathbf{fail}}{\langle P_1 \mid P_2 \sharp v, \sigma\rangle \to_{\mathsf{p}} \mathbf{fail}}$$

## B.4 Dynamic semantics of offers

### B.4.1 Definitions

The dynamic semantics of offers are defined as a relation of the form "$\langle O \sharp v, \sigma\rangle \to_{\mathsf{o}} \sigma'$", where $O$ is an offer, $v$ is a value, and $\sigma$, $\sigma'$ are stores. This relation means that in store $\sigma$, the offer $O$ matches the value $v$, producing the updated store $\sigma'$.

We assume that after binding analysis, named offers have been replaced by positional offers. We thus consider the following abstract syntax of offers:

$$\begin{array}{lll} O & ::= & V \\ & \mid & ?P \end{array}$$

### B.4.2 Send offer

A send offer "$V$" matches a value $v$ only if $v$ is the value of $V$.

$$\frac{\langle V, \sigma\rangle \rightarrow_{\mathsf{e}} v}{\langle V \sharp v, \sigma\rangle \rightarrow_{\mathsf{o}} \sigma}$$

### B.4.3 Receive offer

A receive offer "$?P$" matches a value $v$ only if the pattern $P$ matches $v$.

$$\frac{\langle P \sharp v, \sigma\rangle \rightarrow_{\mathsf{p}} \sigma'}{\langle ?P \sharp v, \sigma\rangle \rightarrow_{\mathsf{o}} \sigma'}$$

## B.5 Dynamic semantics of statements

### B.5.1 Definitions

The dynamic semantics of statements are defined as a relation of the form "$\langle I, \sigma\rangle \xrightarrow{a}_{\mathsf{s}} \sigma'$", where $I$ is a statement, $\sigma$ and $\sigma'$ are stores, and $a$ is a label. This relation means that in store $\sigma$, the statement $I$ terminates, $\sigma'$ being the store obtained after execution of $I$. The label $a$ has one of the following forms:

- "$\surd$" means that $I$ has terminated *normally*. The execution must continue at the next instruction.
- "**brk**$(L)$", where $L$ is the label of a loop, means that $I$ has terminated on a "**break** $L$" statement. The execution must continue at the instruction that follows immediately the loop identified by $L$.
- "**ret**$(v)$" (respectively **ret**), where $v \in \mathcal{V}$, means that $I$ has terminated on a "**return** $v$" (respectively **return**) statement. The execution must continue at the instruction that follows immediately the call to the current function or procedure.

Note that non-terminating statements (e.g., infinite loops or non well-founded recursive functions or procedures) must be considered as incorrect. However, the static semantics cannot guarantee the termination of statements (this problem being undecidable), although it can detect particular cases in which non-termination is certain (see for instance the static semantics rules for the "**loop** $L$" statement in Section 7.11.11). In general, it is the user's responsibility to make sure that statements terminate.

We assume the following:

- After syntactic sugar elimination, array element assignments have been replaced by normal assignment using built-in functions (whose semantics are standard and not given explicitly here) for array update; conditional statements have been replaced by case statements; and all kinds of loops have been replaced by breakable loops.
- After binding analysis, each local variable has been assigned a distinct name, thus enabling local variable declarations to be removed, and named parameter passing has been replaced by positional parameter passing. In addition, for simplicity, we assume that parameters occur in the following order: "**in**" parameters, then "**out**" parameters, then "**in out**" parameters.
- Also, since LNT exceptions are not catchable (for the time being), we do not give rules for exception raising. In practice, raising an exception triggers a runtime error that halts the execution of the LNT specification.
- The LNT construct "**assert** $V$ [**raise** $E$ (...)]" is semantically equivalent to "**if** $V$ **then null else raise** $E$ (...) **end if**". If the "**raise**" clause is missing, then $E$ is taken to be the predefined exception $\xi$ (see Section 6.6) that is implicitly declared at the top level.

We thus consider the following abstract syntax of statements:

$$
\begin{array}{rcl}
I & ::= & \textbf{null} \\
 & | & I_1 \texttt{ ; } I_2 \\
 & | & \textbf{return} \; [\; V \;] \\
 & | & X \texttt{ := } V \\
 & | & [\; X \texttt{ := } ] \; F \; (V_1 \texttt{,} \; \ldots \texttt{,} \; V_m \texttt{,} \; \texttt{?}Y_1 \texttt{,} \; \ldots \texttt{,} \; \texttt{?}Y_p \texttt{,} \; \texttt{!?}Z_1 \texttt{,} \; \ldots \texttt{,} \; \texttt{!?}Z_q) \\
 & | & \textbf{case} \; V \; \textbf{in} \; P_1 \texttt{ -> } I_1 \texttt{ | } \ldots \texttt{ | } P_m \texttt{ -> } I_m \; \textbf{end case} \\
 & | & \textbf{break} \; L \\
 & | & \textbf{loop} \; L \; \textbf{in} \; I_0 \; \textbf{end loop}
\end{array}
$$

## B.5.2 Null

The **null** statement terminates normally and keeps the store unchanged.

$$
\frac{}{\langle \textbf{null}, \sigma \rangle \xrightarrow{\checkmark}_{\mathsf{s}} \sigma}
$$

## B.5.3 Sequential composition

The statement "$I_1$ ; $I_2$" starts by executing $I_1$.

If $I_1$ terminates normally, then $I_2$ is executed in the store updated by $I_1$.

$$\frac{\langle I_1, \sigma\rangle \xrightarrow{\surd}_{\mathsf{s}} \sigma' \quad \langle I_2, \sigma'\rangle \xrightarrow{a}_{\mathsf{s}} \sigma''}{\langle I_1 \ \texttt{;}\ I_2, \sigma\rangle \xrightarrow{a}_{\mathsf{s}} \sigma''}$$

If $I_1$ terminates on a **break** statement or on a **return** statement, then "$I_1 \ \texttt{;}\ I_2$" also terminates on that statement.

$$\frac{\langle I_1, \sigma\rangle \xrightarrow{a}_{\mathsf{s}} \sigma' \quad a \neq \surd}{\langle I_1 \ \texttt{;}\ I_2, \sigma\rangle \xrightarrow{a}_{\mathsf{s}} \sigma'}$$

### B.5.4 Return

A **return** statement terminates, passing a return label to its context.

$$\frac{}{\langle \textbf{return}, \sigma\rangle \xrightarrow{\textbf{ret}}_{\mathsf{s}} \sigma}$$

$$\frac{\langle V, \sigma\rangle \rightarrow_{\mathsf{e}} v}{\langle \textbf{return}\ V, \sigma\rangle \xrightarrow{\textbf{ret}(v)}_{\mathsf{s}} \sigma}$$

### B.5.5 Assignment

An assignment statement terminates normally after updating the store by associating the value of its right-hand side to the assigned variable.

$$\frac{\langle V, \sigma\rangle \rightarrow_{\mathsf{e}} v}{\langle X \ \texttt{:=}\ V, \sigma\rangle \xrightarrow{\surd}_{\mathsf{s}} \sigma \oslash [X \leftarrow v]}$$

### B.5.6 Procedure call that returns a value

Let $F$ be a procedure defined (omitting formal parameter types) as follows:

**function** $F$ **(in** $X_1$**,** $\ldots$**,** $X_m$**, out** $Y_1$**,** $\ldots$**,** $Y_p$**, in out** $Z_1$**,** $\ldots$**,** $Z_q$**)** **:** $T$ **is** $I$ **end function**

A procedure call first evaluates the procedure body $I$ in a store that associates the value of each "**in**" and "**in out**" actual parameter to its respective formal parameter, waiting for a return value $v$ and an updated store $\sigma'$. The execution then terminates normally after updating the initial store, so that the value of each "**out**" and "**in out**" formal parameter in $\sigma'$ is associated to its respective actual parameter, and the return value $v$ is associated to the assigned variable. In the rule below, we use the following abbreviation[1]:

$$\sigma'' \quad \triangleq \quad \sigma \oslash [Y_1' \leftarrow \sigma'(Y_1), \ldots, Y_p' \leftarrow \sigma'(Y_p), Z_1' \leftarrow \sigma'(Z_1), \ldots, Z_q' \leftarrow \sigma'(Z_q), X \leftarrow v]$$

$$\frac{\langle V_i, \sigma\rangle \rightarrow_{\mathsf{e}} v_i \ (i \in 1..m) \quad \left\langle I, [X_1 \leftarrow v_1, \ldots, X_m \leftarrow v_m, Z_1 \leftarrow \sigma(Z_1'), \ldots, Z_q \leftarrow \sigma(Z_q')]\right\rangle \xrightarrow{\mathbf{ret}(v)}_{\mathsf{s}} \sigma'}{\left\langle X \texttt{ := } F \texttt{ (}V_1\texttt{, ..., }V_m\texttt{, ?}Y_1'\texttt{, ..., ?}Y_p'\texttt{, !?}Z_1'\texttt{, ..., !?}Z_q'\texttt{)}, \sigma\right\rangle \xrightarrow{\checkmark}_{\mathsf{s}} \sigma''}$$

### B.5.7 Procedure call that does not return a value

The definition of $F$ has the following form:

**function** $F$ (**in** $X_1$, ..., $X_m$, **out** $Y_1$, ..., $Y_p$, **in out** $Z_1$, ..., $Z_q$) **is** $I$ **end function**

We assume that $I$ necessarily ends with a **return** statement (possibly added by the compiler). In the rule below, we use the following abbreviation:

$$\sigma'' \quad \triangleq \quad \sigma \oslash [Y_1' \leftarrow \sigma'(Y_1), \ldots, Y_p' \leftarrow \sigma'(Y_p), Z_1' \leftarrow \sigma'(Z_1), \ldots, Z_q' \leftarrow \sigma'(Z_q)]$$

$$\frac{\langle V_i, \sigma\rangle \rightarrow_{\mathsf{e}} v_i \ (i \in 1..m) \quad \left\langle I, [X_1 \leftarrow v_1, \ldots, X_m \leftarrow v_m, Z_1 \leftarrow \sigma(Z_1'), \ldots, Z_q \leftarrow \sigma(Z_q')]\right\rangle \xrightarrow{\mathbf{ret}}_{\mathsf{s}} \sigma'}{\left\langle F \texttt{ (}V_1\texttt{, ..., }V_m\texttt{, ?}Y_1'\texttt{, ..., ?}Y_p'\texttt{, !?}Z_1'\texttt{, ..., !?}Z_q'\texttt{)}, \sigma\right\rangle \xrightarrow{\checkmark}_{\mathsf{s}} \sigma''}$$

### B.5.8 Case statement

A case statement "**case** $V$ **in** $P_1$ -> $I_1$ | ... | $P_m$ -> $I_m$ **end case**" first evaluates the value $v$ of the expression $V$. It then executes the first (from left to right) statement $I_i$, whose pattern $P_i$ matches $v$ (if any).

$$\frac{\langle V, \sigma\rangle \rightarrow_{\mathsf{e}} v \quad \langle P_i \sharp v, \sigma\rangle \rightarrow_{\mathsf{p}} \mathbf{fail} \ (i \in 1..j-1) \quad \langle P_j \sharp v, \sigma\rangle \rightarrow_{\mathsf{p}} \sigma_j \quad \langle I_j, \sigma_j\rangle \xrightarrow{a}_{\mathsf{s}} \sigma_j'}{\langle \mathbf{case}\ V\ \mathbf{in}\ P_1 \texttt{ -> } I_1 \texttt{ | ... | } P_m \texttt{ -> } I_m\ \mathbf{end\ case}, \sigma\rangle \xrightarrow{a}_{\mathsf{s}} \sigma_j'} \quad (j \in 1..m)$$

If none of the patterns $P_1, ..., P_m$ matches $v$, then a runtime error (unexpected exception) occurs. This is not explicit in the above rule.

Such a situation may happen because the static semantics do not require case statements to be exhaustive, i.e., $P_1, ..., P_m$ to cover all possible values in the type of $V$. Using non-exhaustive case statements is however unsafe, and signalled at compile-time by a warning message.

[1] The symbol $\triangleq$ should read as *equals by definition*.

### B.5.9 Loop break

A **break** statement terminates, passing the loop label to its context.

$$\frac{}{\langle \textbf{break}\ L, \sigma\rangle \xrightarrow{\textbf{brk}(L)}_{\mathsf{s}} \sigma}$$

### B.5.10 Breakable loop

A breakable loop first executes its body $I_0$.

If $I_0$ terminates normally, then the loop is executed once more in the updated store.

$$\frac{\langle I_0, \sigma\rangle \xrightarrow{\surd}_{\mathsf{s}} \sigma' \quad \langle \textbf{loop}\ L\ \textbf{in}\ I_0\ \textbf{end loop}, \sigma'\rangle \xrightarrow{a}_{\mathsf{s}} \sigma''}{\langle \textbf{loop}\ L\ \textbf{in}\ I_0\ \textbf{end loop}, \sigma\rangle \xrightarrow{a}_{\mathsf{s}} \sigma''}$$

If $I_0$ terminates on a "**break** $L$" statement, $L$ being the label of the current loop, then the loop terminates normally.

$$\frac{\langle I_0, \sigma\rangle \xrightarrow{\textbf{brk}(L)}_{\mathsf{s}} \sigma'}{\langle \textbf{loop}\ L\ \textbf{in}\ I_0\ \textbf{end loop}, \sigma\rangle \xrightarrow{\surd}_{\mathsf{s}} \sigma'}$$

If $I_0$ terminates on a **return** statement or on a "**break** $L'$" statement, $L'$ not being the label of the current loop, then the loop terminates on that statement.

$$\frac{\langle I_0, \sigma\rangle \xrightarrow{a}_{\mathsf{s}} \sigma' \quad a \notin \mathcal{C}_{\mathcal{V}} \quad a \neq \textbf{brk}(L)}{\langle \textbf{loop}\ L\ \textbf{in}\ I_0\ \textbf{end loop}, \sigma\rangle \xrightarrow{a}_{\mathsf{s}} \sigma'}$$

## B.6 Dynamic semantics of behaviours

### B.6.1 Definitions

The dynamic semantics of behaviours are defined as an LTS (*Labeled Transition System*), whose states are couples, often called *configurations* in the literature, of the form "$\langle B, \sigma\rangle$", where $B$ is a behaviour and $\sigma$ is a store. The initial state of an LNT program $B_0$ is "$\langle B_0, []\rangle$". The transitions of the LTS, of the form "$\langle B, \sigma\rangle \xrightarrow{a}_{\mathsf{b}} \langle B', \sigma'\rangle$", are defined by the SOS rules below, where the label $a$ has one of the following forms:

- "$\surd$" (similar to the semantics of statements) is a special label that denotes normal termination. Invariantly, if $a = \surd$ then $B' = \textbf{stop}$, i.e., all transitions labeled by $\surd$ necessarily lead to a deadlock state.
- "**brk**($L$)" (similar to the semantics of statements) is a special label that denotes termination on a "**break** $L$" behaviour. Note that transitions labeled by "**brk**($L$)" can only occur in intermediate SOS steps, and not in the LTS corresponding to the main behaviour of an LNT program.
- A *communication label* has either the form **i** or "$E\ (v_1, \ldots, v_n)$", where $E$ is a gate and $v_1, \ldots, v_n$ are values. We write $\mathcal{C}$ for the set of communication labels. For a communication label $a$, the function gate $(a)$ returns the gate of $a$ as follows:

$$\begin{array}{rcl} \text{gate}\,(\mathbf{i}) & = & \mathbf{i} \\ \text{gate}\,(E\ (v_1, \ldots, v_n)) & = & E \end{array}$$

Note that, unlike statements, behaviours that do not terminate are correct. In general, a non-terminating behaviour produces a potentially infinite sequence of transitions labeled by the communication actions executed along the behaviour execution. As a particular case, a non-terminating behaviour that never reaches any communication action is equivalent to the "**stop**" statement, as it does not produce any transition.

A *gate substitution* is a list of the form "$[E'_1/E_1, \ldots, E'_n/E_n]$", where $E_1, \ldots, E_n, E'_1, \ldots, E'_n$ are gates. We write $\gamma, \gamma_0, \gamma_1, \ldots$ for gate subtitutions. A gate substitution $\gamma = [E'_1/E_1, \ldots, E'_n/E_n]$ can be applied to a behaviour $B$, which is written "$B\gamma$" or "$B[E'_1/E_1, \ldots, E'_n/E_n]$", resulting in the behaviour $B$ in which every occurrence of a gate $E_i$ is replaced by $E'_i$ ($i \in 1..n$). A substitution can alternatively be applied to a label $a$, which is written "$a\gamma$" or "$a[E'_1/E_1, \ldots, E'_n/E_n]$", resulting in the label $a$ whose gate (if any) has been substituted as defined by $\gamma$.

We make the same assumptions as for statements (see Section B.5), and we thus consider the following abstract syntax of behaviours:

$$\begin{array}{rcl}
B & ::= & \textbf{stop} \\
& | & \textbf{null} \\
& | & B_1\ \texttt{;}\ B_2 \\
& | & X\ \texttt{:=}\ V \\
& | & X\ \texttt{:=}\ \textbf{any}\ T\ \textbf{where}\ V \\
& | & [\ X\ \texttt{:=}\ ]\ F\ \texttt{(}V_1\texttt{, ..., }V_m\texttt{, ?}Y_1\texttt{, ..., ?}Y_p\texttt{, !?}Z_1\texttt{, ..., !?}Z_q\texttt{)} \\
& | & \textbf{case}\ V\ \textbf{in}\ P_1\ \texttt{->}\ B_1\ \texttt{| ... |}\ P_m\ \texttt{->}\ B_m\ \textbf{end case} \\
& | & \textbf{break}\ L \\
& | & \textbf{loop}\ L\ \textbf{in}\ B_0\ \textbf{end loop} \\
& | & \Pi\ \texttt{[}E_1\texttt{, ..., }E_n\texttt{] (}V_1\texttt{, ..., }V_m\texttt{, ?}Y_1\texttt{, ..., ?}Y_p\texttt{, !?}Z_1\texttt{, ..., !?}Z_q\texttt{)} \\
& | & E\ \texttt{(}O_1\texttt{, ..., }O_n\texttt{)}\ \textbf{where}\ V \\
& | & \textbf{alt}\ B_1\ \texttt{[]} \ldots \texttt{[]}\ B_n\ \textbf{end alt} \\
& | & \textbf{par}\ E_0, \ldots, E_n\ \textbf{in} \\
& & \quad E_{(1,0)}, \ldots, E_{(1,n_1)}\ \texttt{->}\ B_1\ \texttt{|| ... ||}\ E_{(m,0)}, \ldots, E_{(m,n_m)}\ \texttt{->}\ B_m \\
& & \textbf{end par} \\
& | & \textbf{hide}\ E_1\texttt{, ..., }E_n\ \textbf{in}\ B_0\ \textbf{end hide} \\
& | & \textbf{disrupt}\ B_1\ \textbf{by}\ B_2\ \textbf{end disrupt}
\end{array}$$

### B.6.2 Stop

No SOS rule is associated to $\langle \textbf{stop}, \sigma \rangle$, which represents process inaction.

### B.6.3 Null

The **null** statement terminates normally and keeps the store unchanged.

$$\frac{}{\langle \textbf{null}, \sigma \rangle \xrightarrow{\checkmark}_{\mathsf{b}} \langle \textbf{stop}, \sigma \rangle}$$

### B.6.4 Sequential composition

The behaviour "$B_1$ ; $B_2$" starts by executing $B_1$.
If $B_1$ terminates normally, then $B_2$ is executed in the store updated by $B_1$.

$$\frac{\langle B_1, \sigma \rangle \xrightarrow{\checkmark}_{\mathsf{b}} \langle B_1', \sigma' \rangle \quad \langle B_2, \sigma' \rangle \xrightarrow{a}_{\mathsf{b}} \langle B_2', \sigma'' \rangle}{\langle B_1 \;\texttt{;}\; B_2, \sigma \rangle \xrightarrow{a}_{\mathsf{b}} \langle B_2', \sigma'' \rangle}$$

If $B_1$ terminates on a **break** statement, then "$B_1$ ; $B_2$" also terminates on that statement.

$$\frac{\langle B_1, \sigma \rangle \xrightarrow{\mathbf{brk}(L)}_{\mathsf{b}} \langle B_1', \sigma' \rangle}{\langle B_1 \;\texttt{;}\; B_2, \sigma \rangle \xrightarrow{\mathbf{brk}(L)}_{\mathsf{b}} \langle B_1', \sigma' \rangle}$$

If $B_1$ offers a communication label, then the execution of $B_1$ must continue until termination.

$$\frac{\langle B_1, \sigma \rangle \xrightarrow{a}_{\mathsf{b}} \langle B_1', \sigma' \rangle \quad a \in \mathcal{C}}{\langle B_1 \;\texttt{;}\; B_2, \sigma \rangle \xrightarrow{a}_{\mathsf{b}} \langle B_1' \;\texttt{;}\; B_2, \sigma' \rangle}$$

### B.6.5 Deterministic assignment

A deterministic assignment terminates normally after updating the store by associating the value of its right-hand side to the assigned variable.

$$\frac{\langle V, \sigma \rangle \rightarrow_{\mathsf{e}} v}{\langle X \;\texttt{:=}\; V, \sigma \rangle \xrightarrow{\checkmark}_{\mathsf{b}} \langle \textbf{stop}, \sigma \oslash [X \leftarrow v] \rangle}$$

### B.6.6 Nondeterministic assignment

A nondeterministic assignment terminates normally after updating the store by associating a value to the assigned variable, provided the condition of the assignment evaluates to **true** in the updated store.

$$\frac{v \in T \quad \sigma' = \sigma \oslash [X \leftarrow v] \quad \langle V, \sigma' \rangle \to_{\mathsf{e}} \textbf{true}}{\langle X \texttt{ := } \textbf{any } T \textbf{ where } V, \sigma \rangle \xrightarrow{\surd}_{\mathsf{b}} \langle \textbf{stop}, \sigma' \rangle}$$

### B.6.7 Procedure call that returns a value

The behaviour semantics of such a procedure call are directly derived from its statement semantics.

$$\frac{\langle X \texttt{ := } F\ \texttt{(}V_1\texttt{,}\ \ldots\texttt{,}\ V_m\texttt{,}\ \texttt{?}Y_1\texttt{,}\ \ldots\texttt{,}\ \texttt{?}Y_p\texttt{,}\ \texttt{!?}Z_1\texttt{,}\ \ldots\texttt{,}\ \texttt{!?}Z_q\texttt{)}, \sigma \rangle \xrightarrow{\surd}_{\mathsf{s}} \sigma'}{\langle X \texttt{ := } F\ \texttt{(}V_1\texttt{,}\ \ldots\texttt{,}\ V_m\texttt{,}\ \texttt{?}Y_1\texttt{,}\ \ldots\texttt{,}\ \texttt{?}Y_p\texttt{,}\ \texttt{!?}Z_1\texttt{,}\ \ldots\texttt{,}\ \texttt{!?}Z_q\texttt{)}, \sigma \rangle \xrightarrow{\surd}_{\mathsf{b}} \langle \textbf{stop}, \sigma' \rangle}$$

### B.6.8 Procedure that does not return a value

The behaviour semantics of such a procedure call are directly derived from its statement semantics.

$$\frac{\langle F\ \texttt{(}V_1\texttt{,}\ \ldots\texttt{,}\ V_m\texttt{,}\ \texttt{?}Y_1\texttt{,}\ \ldots\texttt{,}\ \texttt{?}Y_p\texttt{,}\ \texttt{!?}Z_1\texttt{,}\ \ldots\texttt{,}\ \texttt{!?}Z_q\texttt{)}, \sigma \rangle \xrightarrow{\surd}_{\mathsf{s}} \sigma'}{\langle F\ \texttt{(}V_1\texttt{,}\ \ldots\texttt{,}\ V_m\texttt{,}\ \texttt{?}Y_1\texttt{,}\ \ldots\texttt{,}\ \texttt{?}Y_p\texttt{,}\ \texttt{!?}Z_1\texttt{,}\ \ldots\texttt{,}\ \texttt{!?}Z_q\texttt{)}, \sigma \rangle \xrightarrow{\surd}_{\mathsf{b}} \langle \textbf{stop}, \sigma' \rangle}$$

Note that in both rules above, $\surd$ is the only label that can possibly be obtained from the statement semantics of a procedure call.

### B.6.9 Case behaviour

The dynamic semantics of a case behaviour are similar to the dynamic semantics of a case statement.

$$\frac{\langle V, \sigma \rangle \to_{\mathsf{e}} v \quad \langle P_i \sharp v, \sigma \rangle \to_{\mathsf{p}} \textbf{fail}\ (i \in 1..j-1) \quad \langle P_j \sharp v, \sigma \rangle \to_{\mathsf{p}} \sigma_j \quad \langle B_j, \sigma_j \rangle \xrightarrow{a}_{\mathsf{b}} \langle B'_j, \sigma'_j \rangle}{\langle \textbf{case } V \textbf{ in } P_1 \texttt{ -> } B_1 \texttt{ | } \ldots \texttt{ | } P_m \texttt{ -> } B_m \textbf{ end case}, \sigma \rangle \xrightarrow{a}_{\mathsf{b}} \langle B'_j, \sigma'_j \rangle} \quad (j \in 1..m)$$

If none of the patterns $P_1, ..., P_m$ matches $v$, then a runtime error (unexpected exception) occurs. This is not explicit in the above rule. See discussion about case exhaustivity in Section B.5.8.

### B.6.10 Loop break

A **break** behaviour terminates, passing the loop label to its context.

$$\frac{}{\langle \textbf{break}\ L, \sigma\rangle \xrightarrow{\textbf{brk}(L)}_{\mathsf{b}} \langle \textbf{stop}, \sigma\rangle}$$

### B.6.11 Breakable loop

The dynamic semantics of a breakable loop behaviour are slightly more complicated than those of a breakable loop statement, because of the possible occurrence of an unknown number of communications in the loop body before termination. The introduction of an intermediate construct written "*loop* $(L, B_1, B_2)$" is necessary. Its semantics are defined as follows.

The behaviour "*loop* $(L, B_1, B_2)$" starts by executing $B_1$.

If $B_1$ offers a communication label then *loop* $(L, B_1, B_2)$ offers this communication label.

$$\frac{\langle B_1, \sigma\rangle \xrightarrow{a}_{\mathsf{b}} \langle B_1', \sigma'\rangle \qquad a \in \mathcal{C}}{\langle \mathit{loop}\,(L, B_1, B_2), \sigma\rangle \xrightarrow{a}_{\mathsf{b}} \langle \mathit{loop}\,(L, B_1', B_2), \sigma'\rangle}$$

If $B_1$ terminates on a "**break** $L$" statement, then *loop* $(L, B_1, B_2)$ terminates normally.

$$\frac{\langle B_1, \sigma\rangle \xrightarrow{\textbf{brk}(L)}_{\mathsf{b}} \langle B_1', \sigma'\rangle}{\langle \mathit{loop}\,(L, B_1, B_2), \sigma\rangle \xrightarrow{\surd}_{\mathsf{b}} \langle B_1', \sigma'\rangle}$$

If $B_1$ terminates on a "**break** $L'$" statement, where $L' \neq L$, then *loop* $(L, B_1, B_2)$ terminates on that statement.

$$\frac{\langle B_1, \sigma\rangle \xrightarrow{\textbf{brk}(L')}_{\mathsf{b}} \langle B_1', \sigma'\rangle \qquad L' \neq L}{\langle \mathit{loop}\,(L, B_1, B_2), \sigma\rangle \xrightarrow{\textbf{brk}(L')}_{\mathsf{b}} \langle B_1', \sigma'\rangle}$$

If $B_1$ terminates normally (without a **break**), then $B_2$ is executed in the store updated by $B_1$.

$$\frac{\langle B_1, \sigma\rangle \xrightarrow{\surd}_{\mathsf{b}} \langle B_1', \sigma'\rangle \qquad \langle B_2, \sigma'\rangle \xrightarrow{a}_{\mathsf{b}} \langle B_2', \sigma''\rangle}{\langle \mathit{loop}\,(L, B_1, B_2), \sigma\rangle \xrightarrow{a}_{\mathsf{b}} \langle B_2', \sigma''\rangle}$$

The SOS for a breakable loop are therefore given by the following single rule.

$$\frac{\langle \mathit{loop}\,(L, B_0, \textbf{loop}\ L\ \textbf{in}\ B_0\ \textbf{end loop}), \sigma\rangle \xrightarrow{a}_{\mathsf{b}} \langle B_0', \sigma'\rangle}{\langle \textbf{loop}\ L\ \textbf{in}\ B_0\ \textbf{end loop}, \sigma\rangle \xrightarrow{a}_{\mathsf{b}} \langle B_0', \sigma'\rangle}$$

### B.6.12 Process call

Although the static semantics currently restrict process recursion to tail recursion, the following Sos rules encompass the case of general recursion, which might become available in future versions of Lnt. To this end, we introduce an intermediate behaviour construct called a *closure*, denoted by "$\mathit{call}\,(B, I, \gamma, \sigma)$", where $B$ is a behaviour, $I$ is a (possibly empty) sequence of assignments of the form "$X_1'$ `:=` $X_1$ `;` ... `;` $X_m'$ `:=` $X_m$" ($m \geq 0$), $\gamma$ is a gate substitution of the form "$[E_1'/E_1, \ldots, E_n'/E_n]$" ($n \geq 0$), and $\sigma$ is a store.

We assume that $\Pi$ is a process defined (omitting formal parameter types) as follows:

**process** $\Pi$ `[` $E_1$ `,` ... `,` $E_n$ `]` `(`**in** $X_1$ `,` ... `,` $X_m$ `,` **out** $Y_1$ `,` ... `,` $Y_p$ `,` **in out** $Z_1$ `,` ... `,` $Z_q$`)` **is**
  $B$
**end process**

When calling process $\Pi$, a closure is created, containing the process body $B$, a sequence of assignments $I$ implementing the update of "**out**" and "**in out**" parameters, a gate substitution $\gamma$ implementing gate parameter passing, and the current store $\sigma$, which is the store of the caller. At the same time, a local store $\sigma'$ is created, assigning the values of "**in**" and "**in out**" actual parameters to the corresponding formal parameters. The closure is then executed in the local store $\sigma'$, which is the store of the callee. In the rule below, we use the following abbreviations:

$$\begin{array}{rcl}
\gamma & \triangleq & [E_1'/E_1, \ldots, E_n'/E_n] \\
I & \triangleq & Y_1' \texttt{ := } Y_1 \texttt{ ; } \ldots \texttt{ ; } Y_p' \texttt{ := } Y_p \texttt{ ; } Z_1' \texttt{ := } Z_1 \texttt{ ; } \ldots \texttt{ ; } Z_q' \texttt{ := } Z_q \\
\sigma' & \triangleq & [X_1 \leftarrow v_1, \ldots, X_m \leftarrow v_m, Z_1 \leftarrow \sigma(Z_1'), \ldots, Z_q \leftarrow \sigma(Z_q')]
\end{array}$$

$$\frac{\langle V_i, \sigma\rangle \rightarrow_{\mathsf{e}} v_i\ \ (i \in 1..m) \qquad\qquad \langle \mathit{call}\,(B, I, \gamma, \sigma), \sigma'\rangle \xrightarrow{a}_{\mathsf{b}} \langle B'', \sigma''\rangle}{\left\langle \Pi\ \texttt{[}E_1'\texttt{,}\ \ldots\texttt{,}\ E_n'\texttt{]}\ \texttt{(}V_1\texttt{,}\ \ldots\texttt{,}\ V_m\texttt{,}\ \texttt{?}Y_1'\texttt{,}\ \ldots\texttt{,}\ \texttt{?}Y_p'\texttt{,}\ \texttt{!?}Z_1'\texttt{,}\ \ldots\texttt{,}\ \texttt{!?}Z_q'\texttt{)}, \sigma\right\rangle \xrightarrow{a}_{\mathsf{b}} \langle B'', \sigma''\rangle}$$

If the body of the process offers a communication label, then the communication label is renamed according to gate parameters. The execution then continues normally.

$$\frac{\langle B, \sigma\rangle \xrightarrow{a}_{\mathsf{b}} \langle B', \sigma'\rangle \qquad\qquad a \in \mathcal{C}}{\langle \mathit{call}\,(B, I, \gamma, \sigma_0), \sigma\rangle \xrightarrow{a\gamma}_{\mathsf{b}} \langle \mathit{call}\,(B', I, \gamma, \sigma_0), \sigma'\rangle}$$

If the body of the process terminates normally, then the process call also terminates normally after restoring the store of the caller and updating the "**out**" and "**in out**" actual parameters.

$$\frac{\langle B, \sigma\rangle \xrightarrow{\checkmark}_{\mathsf{b}} \langle B', \sigma'\rangle}{\langle \mathit{call}\,(B, \{X_1' \texttt{ := } X_1 \texttt{ ; } \ldots \texttt{ ; } X_n' \texttt{ := } X_n\}, \gamma, \sigma_0), \sigma\rangle \xrightarrow{\checkmark}_{\mathsf{b}} \langle B', \sigma_0 \oslash [X_1' \leftarrow \sigma'(X_1), \ldots, X_n' \leftarrow \sigma'(X_n)]\rangle}$$

Note that the static semantics ensure that no transition labeled by "**brk**($L$)" can be derived from a closure, because a **break** behaviour can only interrupt a loop that belongs to the process body.

### B.6.13 Communication

If $v_1, \ldots, v_n$ are (nondeterministic) values matching the offers $O_1, \ldots, O_n$ in such a way that the guard $V$ evaluates to **true**, then the communication behaviour "$E$ `(`$O_1$`, ..., `$O_n$`)` **where** $V$" offers the communication label "$E\ (v_1, \ldots, v_n)$" and then behaves like **null**, so as to enable execution of the next behaviour.

$$\frac{\sigma_0 = \sigma \quad \langle O_i \sharp v_i, \sigma_{i-1}\rangle \rightarrow_{\mathsf{o}} \sigma_i\ (i \in 1..n) \quad \langle V, \sigma'\rangle \rightarrow_{\mathsf{e}} \mathbf{true}}{\langle E\ \texttt{(}O_1\texttt{, ..., }O_n\texttt{)}\ \mathbf{where}\ V, \sigma\rangle \xrightarrow{E\ (v_1,\ldots,v_n)}_{\mathsf{b}} \langle \mathbf{null}, \sigma_n\rangle}$$

The above rule ensure that offers are evaluated from left to right. Theoretically, this would allow a variable bound in an offer to be used in an expression (e.g., a Boolean condition) located further to the right, although for practical reasons, this is not currently allowed by the static semantics.

### B.6.14 Nondeterministic choice

A nondeterministic choice between behaviours $B_1, \ldots, B_n$ behaves as any of the $B_i$ behaviours.

$$\frac{\langle B_i, \sigma\rangle \xrightarrow{a}_{\mathsf{b}} \langle B_i', \sigma'\rangle}{\langle \mathbf{alt}\ B_1\ \texttt{[]} \ldots \texttt{[]}\ B_n\ \mathbf{end\ alt}, \sigma\rangle \xrightarrow{a}_{\mathsf{b}} \langle B_i', \sigma'\rangle} \quad (i \in 1..n)$$

### B.6.15 Parallel composition

In the first SOS rule below, for a communication label $a \in \mathcal{C}$, sync$(a)$ denotes a set of subsets of $1..m$, each such subset (called a *synchronization set*) denoting the indices of the behaviours among $B_1, \ldots, B_m$ that synchronize on $a$. It is defined as follows:

$$\text{sync}\,(a) = \begin{cases} \{\{1..m\}\} & \text{if gate}\,(a) \in \{E_0, \ldots, E_n\} \\ \{\{i \mid i \in 1..m \wedge \text{gate}\,(a) \in \{E_{(i,0)}, \ldots, E_{(i,n_i)}\}\}\} & \text{if gate}\,(a) \in \bigcup_{i \in 1..m}\{E_{(i,0)}, \ldots, E_{(i,n_i)}\} \\ \{\{i\} \mid i \in 1..m\} & \text{otherwise} \end{cases}$$

Note that the static semantics ensure that all three cases in the definition above are exclusive.

If $S$ is a synchronization set for a communication label $a$, and if each behaviour in $S$ offers $a$ while the behaviours outside $S$ remain idle, then the parallel composition offers $a$. In the rule below, we use the following abbreviation:

$$\sigma' \triangleq \sigma \oslash ((\sigma_1 \ominus \sigma) \oplus \ldots \oplus (\sigma_m \ominus \sigma))$$

$$\frac{a \in \mathcal{C} \quad S \in \text{sync}\,(a) \quad \langle B_i, \sigma \rangle \xrightarrow{a}_{\mathsf{b}} \langle B'_i, \sigma_i \rangle \;\; (i \in S) \quad \left\langle B'_j, \sigma_j \right\rangle = \langle B_j, \sigma \rangle \;\; (j \in 1..m \setminus S)}{\left\langle \begin{array}{l} \textbf{par } E_0, \ldots, E_n \textbf{ in} \\ \quad E_{(1,0)}, \ldots, E_{(1,n_1)} \texttt{ -> } B_1 \\ \quad \texttt{|| ... ||} \\ \quad E_{(m,0)}, \ldots, E_{(m,n_m)} \texttt{ -> } B_m \\ \textbf{end par} \end{array}, \sigma \right\rangle \xrightarrow{a}_{\mathsf{b}} \left\langle \begin{array}{l} \textbf{par } E_0, \ldots, E_n \textbf{ in} \\ \quad E_{(1,0)}, \ldots, E_{(1,n_1)} \texttt{ -> } B'_1 \\ \quad \texttt{|| ... ||} \\ \quad E_{(m,0)}, \ldots, E_{(m,n_m)} \texttt{ -> } B'_m \\ \textbf{end par} \end{array}, \sigma' \right\rangle}$$

If all parallel behaviours terminate normally, then the parallel composition terminates normally. Again in the rule below, we use the following abbreviation:

$$\sigma' \triangleq \sigma \oslash ((\sigma_1 \ominus \sigma) \oplus \ldots \oplus (\sigma_m \ominus \sigma))$$

$$\frac{\langle B_i, \sigma \rangle \xrightarrow{\surd}_{\mathsf{b}} \langle B'_i, \sigma_i \rangle \;\; (i \in 1..m)}{\left\langle \begin{array}{l} \textbf{par } E_0, \ldots, E_n \textbf{ in} \\ \quad E_{(1,0)}, \ldots, E_{(1,n_1)} \texttt{ -> } B_1 \\ \quad \texttt{|| ... ||} \\ \quad E_{(m,0)}, \ldots, E_{(m,n_m)} \texttt{ -> } B_m \\ \textbf{end par} \end{array}, \sigma \right\rangle \xrightarrow{\surd}_{\mathsf{b}} \langle \textbf{stop}, \sigma' \rangle}$$

In both rules above, the resulting store $\sigma'$ is the initial store updated with respect to the union of store updates performed locally in the parallel branches. Note that the static semantics ensure that the sets of variables on which the stores $\sigma_1, \ldots, \sigma_m$ are defined are mutually disjoint, because each variable can be updated in at most one parallel branch. Hence, the store $(\sigma_1 \ominus \sigma) \oplus \ldots \oplus (\sigma_m \ominus \sigma)$ is well-defined. Also, the order of $B_1, \ldots, B_m$ in the parallel composition has no effect on the resulting store as disjoint union is associative and commutative.

Sos rules for parallel behaviours terminating on a **break** behaviour are unnecessary because the static semantics ensure that if one of the parallel behaviours executes a **break** then the broken loop also occurs in the same parallel behaviour. Therefore, every label of the form "**brk**($L$)" has necessarily already been turned into a $\surd$ by the Sos rule for breakable loops.

## B.6.16 Hiding

If the body of a **hide** behaviour offers a communication label whose gate belongs to the set of gates to be hidden, then the communication label offered by the **hide** behaviour is the internal action "**i**".

$$\frac{\langle B_0, \sigma \rangle \xrightarrow{a}_{\mathsf{b}} \langle B'_0, \sigma' \rangle \qquad a \in \mathcal{C} \qquad \text{gate}\,(a) \in \{E_1, \ldots, E_n\}}{\langle \textbf{hide } E_1\textbf{,} \ldots\textbf{,} \ E_n \textbf{ in } B_0 \textbf{ end hide}, \sigma \rangle \xrightarrow{\mathbf{i}}_{\mathsf{b}} \langle \textbf{hide } E_1\textbf{,} \ldots\textbf{,} \ E_n \textbf{ in } B'_0 \textbf{ end hide}, \sigma' \rangle}$$

If the body of the **hide** behaviour offers a communication label whose gate does not belong to the set of gates to be hidden, then the **hide** behaviour offers this communication label.

$$\frac{\langle B_0, \sigma\rangle \xrightarrow{a}_{\mathtt{b}} \langle B_0', \sigma'\rangle \qquad a \in \mathcal{C} \qquad \mathrm{gate}\,(a) \notin \{E_1, \ldots, E_n\}}{\langle \textbf{hide } E_1\textbf{, } \ldots\textbf{, } E_n \textbf{ in } B_0 \textbf{ end hide}, \sigma\rangle \xrightarrow{a}_{\mathtt{b}} \langle \textbf{hide } E_1\textbf{, } \ldots\textbf{, } E_n \textbf{ in } B_0' \textbf{ end hide}, \sigma'\rangle}$$

If the body of the **hide** behaviour terminates, then the **hide** behaviour also terminates.

$$\frac{\langle B_0, \sigma\rangle \xrightarrow{a}_{\mathtt{b}} \langle B_0', \sigma'\rangle \qquad a \notin \mathcal{C}}{\langle \textbf{hide } E_1\textbf{, } \ldots\textbf{, } E_n \textbf{ in } B_0 \textbf{ end hide}, \sigma\rangle \xrightarrow{a}_{\mathtt{b}} \langle B_0', \sigma'\rangle}$$

### B.6.17 Disrupting

If the left-hand behaviour of a **disrupt** behaviour offers a communication label, then the **disrupt** behaviour also offers this communication label, without disabling its right-hand behaviour.

$$\frac{\langle B_1, \sigma\rangle \xrightarrow{a}_{\mathtt{b}} \langle B_1', \sigma'\rangle \qquad a \in \mathcal{C}}{\langle \textbf{disrupt } B_1 \textbf{ by } B_2 \textbf{ end disrupt}, \sigma\rangle \xrightarrow{a}_{\mathtt{b}} \langle \textbf{disrupt } B_1' \textbf{ by } B_2 \textbf{ end disrupt}, \sigma'\rangle}$$

If the left-hand behaviour of the **disrupt** behaviour terminates normally or on a **break** behaviour, then the **disrupt** behaviour also terminates.

$$\frac{\langle B_1, \sigma\rangle \xrightarrow{a}_{\mathtt{b}} \langle B_1', \sigma'\rangle \qquad a \notin \mathcal{C}}{\langle \textbf{disrupt } B_1 \textbf{ by } B_2 \textbf{ end disrupt}, \sigma\rangle \xrightarrow{a}_{\mathtt{b}} \langle B_1', \sigma'\rangle}$$

Finally, at any time, the **disrupt** behaviour may behave as its right-hand behaviour, thus disabling its left-hand behaviour.

$$\frac{\langle B_2, \sigma\rangle \xrightarrow{a}_{\mathtt{b}} \langle B_2', \sigma'\rangle}{\langle \textbf{disrupt } B_1 \textbf{ by } B_2 \textbf{ end disrupt}, \sigma\rangle \xrightarrow{a}_{\mathtt{b}} \langle B_2', \sigma'\rangle}$$

## B.7 Discussion on the dynamics semantics

LNT dynamic semantics are defined formally in Appendix B. Note that LNT relies on a notion of semantic equivalence (namely, strong bisimulation) that gives an account of the branching structure of programs. This implies that behaviour executions must be thought of as trees rather than traces, i.e., the locations where nondeterministic choices are resolved during program execution are meaningful. Therefore, reasoning about LNT program equivalences is more subtle than standard (generally trace-based) program equivalences.

To illustrate this, consider a sequential composition "$B_1$ ; $B_2$ ; $B_3$", such that communication occurs in $B_1$ and $B_3$ but not in $B_2$. If every variable used in $B_1$ is not modified in $B_2$ and conversely, one might think that "$B_1$ ; $B_2$ ; $B_3$" and "$B_2$ ; $B_1$ ; $B_3$" denote equivalent behaviours. In fact, this is not true if $B_2$ is nondeterministic.

More concretely, let $B_1$ be "$E_0$" (communication on gate $E_0$ without offers), $B_2$ be the nondeterministic assignment "$b$ **:= any bool**", and $B_3$ be defined by "**if** $b$ **then** $E_1$ **else** $E_2$ **end if**". Then, the programs "$B_1$ ; $B_2$ ; $B_3$" and "$B_2$ ; $B_1$ ; $B_3$" are not equivalent:

- In "$B_1$ ; $B_2$ ; $B_3$" (i.e., "$E_0$ ; $b$ **:= any bool; if** $b$ **then** $E_1$ **else** $E_2$ **end if**"), $E_0$ is first executed deterministically, leading the program to a state in which there is a nondeterministic choice between $E_1$ and $E_2$.
- In "$B_2$ ; $B_1$ ; $B_3$" (i.e., "$b$ **:= any bool;** $E_0$ **; if** $b$ **then** $E_1$ **else** $E_2$ **end if**"), there is a nondeterministic choice on $E_0$ initially: the program may either execute $E_0$ then $E_1$ (if $b$ is **true**), or execute $E_0$ then $E_2$ (otherwise), but there is no state in which the program has a choice between $E_1$ and $E_2$.

In general, it is recommended to think carefully about the order in which communications and nondeterministic behaviours should be combined.

# Appendix C

# Predefined functions

This appendix lists the predefined functions that can be used in an LNT program over the six basic types (Booleans, natural numbers, integers, real numbers, characters, and strings). The LOTOS code for these functions is defined in the "`LNT_V1.lib`" file and the files it includes.

Every predefined LNT function named $F$ is implemented in "`LNT_V1.lib`" by a LOTOS operation that is also named $F$, except `=` (which is implemented by `==`) and `!=` (which is implemented by `/=`), because `=` and `!=` are not valid LOTOS operation names.

In each section, the table shows the predefined functions. Binary functions can be used in either prefix or infix mode, whereas other functions can be used in infix mode only.

The predefined functions over non-basic types (e.g., list, sorted list, and set types) are defined in Chapter 5.

In addition to these predefined functions, a set of predefined libraries can be found in the directory "`$LNT_LOCATION/lib`" (look for files having the "`.lnt`" extension). The data types and associated functions provided by these libraries can be consulted by reading the corresponding LNT code.

## C.1 Functions on Booleans

| Functions | Profile |
|---|---|
| `and`, `and then`, `or`, `or else`, `xor`, `=>`, `<=>` | Bool, Bool → Bool |
| `==`, `=`, `<>`, `!=`, `<`, `<=`, `>`, `>=` | Bool, Bool → Bool |
| `false`, `true` | → Bool |
| `not` | Bool → Bool |
| `Succ`, `Pred` | Bool → Bool |
| `String` | Bool → String |

## C.2 Functions on natural numbers

| Functions | Profile |
|---|---|
| `+`, `-`, `*`, `**`, `div`, `mod`, `min`, `max`, `gcd`, `scm` | Nat, Nat → Nat |
| `==`, `=`, `<>`, `!=`, `<`, `<=`, `>`, `>=` | Nat, Nat → Bool |
| `Succ`, `Pred` | Nat → Nat |
| `Char` | Nat → Char |
| `Int` | Nat → Int |
| `Real` | Nat → Real |
| `String` | Nat → String |

## C.3 Functions on integer numbers

| Functions | Profile |
|---|---|
| `+`, `-` (minus), `*`, `div`, `rem`, `mod` | Int, Int → Int |
| `**` | Int, Nat → Int |
| `==`, `=`, `<>`, `!=`, `<`, `<=`, `>`, `>=` | Int, Int → Bool |
| `min`, `max` | Int, Int → Int |
| `Pos`, `Neg` | Nat → Int |
| `+` (opposite), `-` (opposite) | Int → Int |
| `Succ`, `Pred`, `sign`, `abs` | Int → Int |
| `Nat` | Int → Nat |
| `Real` | Int → Real |
| `String` | Int → String |

Note: Functions `rem` and `mod` denote respectively the *remainder* and the *modulo* of two integer numbers:

- The definition of `rem` is consistent with the mathematical definition of remainder in Euclidian division, satisfying the law `x rem y = x - (y * (x div y))`. The result is equal to zero or has the same sign as the left operand.

- The definition of `mod` is consistent with the mathematical definition of the modulo operator in modulo arithmetic, satisfying the law `(x+n) mod n = x mod n`. The result is equal to zero or has the same sign as the right operand.

Both functions coincide if both operands have the same sign or if the left operand is a multiple of the right operand. They may yield different results in all other cases.

## C.4 Functions on real numbers

| Functions | Profile |
|---|---|
| `==`, `=`, `<>`, `!=`, `<`, `<=`, `>`, `>=` | Real, Real → Bool |
| `+`, `-` (minus), `*`, `/`, `**` | Real, Real → Real |
| `-` (opposite) | Real → Real |
| `abs` | Real → Real |
| `Int` | Real → Int |
| `String` | Real → String |

## C.5 Functions on characters

| Functions | Profile |
|---|---|
| `==`, `=`, `<>`, `!=`, `<`, `<=`, `>`, `>=` | Char, Char → Bool |
| `IsLower`, `IsUpper`, `IsAlpha`, `IsAlnum`, `IsDigit`, `IsXDigit` | Char → Bool |
| `ToLower`, `ToUpper` | Char → Char |
| `Succ`, `Pred` | Char → Char |
| `Nat` | Char → Nat |
| `String` | Char → String |

## C.6 Functions on strings

| Functions | Profile |
|---|---|
| `&`, `~` | String, String → String |
| `prefix`, `suffix` | String, Nat → String |
| `element` | String, Nat → Char |
| `index`, `rindex` | String, String → Nat |
| `==`, `=`, `<>`, `!=`, `<`, `<=`, `>`, `>=` | String, String → Bool |
| `length` | String → Nat |
| `empty` | String → Bool |
| `substr` | String, Nat, Nat → String |
| `Char` | String → Char |
| `Nat` | String → Nat |
| `Int` | String → Int |
| `Real` | String → Real |
| `String` | String → String |

# Appendix D

# Examples

## D.1 LNT types

### D.1.1 Enumerated type

Here is an example which defines a simple enumerated LNT data type WEEK_DAY:

```
module DAY is
   type WEEK_DAY is
      MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
      with ==, !=
   end type
end module
```

The definition of the LNT type takes only 6 lines. If the same type had been written in LOTOS, it would have taken 20 lines. Here is an idea of the LOTOS code generated by LNT2LOTOS for this example:

```
type DAY is
   sorts WEEK_DAY
opns
   MONDAY    (*! constructor *) : -> WEEK_DAY
   TUESDAY   (*! constructor *) : -> WEEK_DAY
   WEDNESDAY (*! constructor *) : -> WEEK_DAY
   THURSDAY  (*! constructor *) : -> WEEK_DAY
   FRIDAY    (*! constructor *) : -> WEEK_DAY
   SATURDAY  (*! constructor *) : -> WEEK_DAY
   SUNDAY    (*! constructor *) : -> WEEK_DAY

   _==_ : WEEK_DAY, WEEK_DAY -> BOOL
   _/=_ : WEEK_DAY, WEEK_DAY -> BOOL
eqns
   forall x, y : WEEK_DAY
      ofsort BOOL
         x == x = true;
         x == y = false;
      ofsort BOOL
         x /= y = not (x == y);
```

```
endtype
```

### D.1.2 Record type

This section gives an example of a record type. The PERSON type stores information about a person.

The type NAT is assumed to be defined in a module called NATURAL and represents the natural numbers. The type STRING is defined in a module called STRING and represents character strings.

```
module PERSON (NATURAL, STRING) is
   type GENDER is
      F, M
   end type

   type PERSON is
      PERSON (NAME : STRING, SURNAME : STRING, AGE : NAT, SEX : GENDER)
   end type
end module
```

The corresponding generated LOTOS code is:

```
type PERSON is NATURAL, STRING
   sorts
      GENDER,
      PERSON
   opns
      (* constructors for sort "GENDER" *)
      F (*! constructor *) :  -> GENDER
      M (*! constructor *) :  -> GENDER

      (* constructors for sort "PERSON" *)
      PERSON (*! constructor *) : STRING, STRING, NAT, GENDER -> PERSON
endtype
```

### D.1.3 List type

A list of booleans could be defined as follows:

```
module BOOLEAN_LIST (BOOLEAN) is
   type BOOLEAN_LIST is
      list of BOOL
   end type
end module
```

This is a shorthand notation to define a type with two constructors CONS and NIL. The following piece of LNT code defines exactly the same type:

```
module BOOLEAN_LIST (BOOLEAN) is
   type BOOLEAN_LIST is
      NIL,
      CONS (HEAD : BOOL, TAIL : BOOLEAN_LIST)
   end type
end module
```

The corresponding generated LOTOS code is:

```
type BOOLEAN_LIST is BOOLEAN
   sorts BOOLEAN_LIST
   opns
      NIL (*! constructor *) :   -> BOOLEAN_LIST
      CONS (*! constructor *) : BOOL, BOOLEAN_LIST -> BOOLEAN_LIST
endtype
```

### D.1.4 Array types

An array of three natural numbers could be defined as follows:

```
type Nat_Array is
  array [0 .. 2] of Nat
end type
```

An array of a records containing a pair of natural numbers could be defined as follows:

```
type Record is
  Record (n, m: Nat)
with get, set
end type

type Record_Array is
  array [0 .. 1] of Record
end type
```

An array of arrays of natural numbers could be defined as follows:

```
type Nat_Array is
  array [0 .. 1] of Nat
end type

type Nat_Array_Array is
  array [0 .. 1] of Nat_Array
end type
```

## D.2 LNT functions

### D.2.1 Manipulating record fields

Consider the following nested record types:

```
module PERSON (NATURAL, STRING) with get, set is
   type GENDER is
      F, M
   end type

   type NAME is
      NAME (FIRST_NAME, LAST_NAME : STRING)
   end type
```

```
    type PERSON is
      PERSON (NAME: NAME, AGE: NAT, SEX: GENDER)
    end type
  end module
```

The following two functions illustrate the use of field updates (see Section 7.13.6) to change fields of a (nested) record:

```
  function CHANGE_AGE (in out P: PERSON, NEW_AGE: NAT) is
    P := P.{AGE -> NEW_AGE}
  end function

  function CHANGE_LAST_NAME (in out P: PERSON, NEW_LAST_NAME: STRING) is
    P := P.{NAME -> P.NAME.{LAST_NAME -> NEW_LAST_NAME}}
  end function
```

### D.2.2 The factorial function

The following example gives several implementations of the factorial function, and shows how to use the main LNT features.

```
module FACT (NATURAL) is
  (* while loop *)
  function FACT1 (N : NAT) : NAT is
    var RESULT : NAT := 1,
        I : NAT := 1
    in
      while I <= N loop
        RESULT := RESULT * I;
        I := I + 1
      end loop;
      return RESULT
    end var
  end function

  (* for loop *)
  function FACT2 (N : NAT) : NAT is
    var RESULT, I : NAT in
      RESULT := 1;
      for I := 1 while I <= N by I := I + 1 loop
        RESULT := RESULT * I;
      end loop;
     return RESULT
    end var
  end function

  (* breakable loop *)
  function FACT3 (N : NAT) : NAT is
    var RESULT, I : NAT in
      RESULT := 1;
      I := 1;
      loop L in
```

```
          if I > N then
             break L
          end if;
          RESULT := RESULT * I;
          I := I + 1
       end loop;
       return RESULT
    end var
 end function

 (* recursive *)
 function FACT4 (N : NAT) : NAT is
    if N == 0 then
       return 1
    else
       return N * FACT4 (N - 1)
    end if
 end function

 (* another recursive *)
 function FACT5 (N : NAT) : NAT is
    case N of NAT
    var I : NAT in
       0 -> return 1
     | I -> return I * FACT5 (I - 1)
    end case
 end function

 (* tail-recursive *)
 function FACT6 (N : NAT) : NAT is
    return FACT6 (N, 1)
 end function

 function FACT6 (N, ACC : NAT) : NAT is
    if N == 0 then
       return ACC
    else
       return FACT6 (N - 1, ACC * N)
    end if
 end function

 (* another tail-recursive *)
 function FACT7 (N : NAT) : NAT is
    return FACT7 (N, 1)
 end function

 function FACT7 (N, ACC : NAT) : NAT is
    case N of NAT
    var I : NAT in
       0 -> return ACC
     | I -> return FACT7 (I - 1, ACC * I)
    end case
 end function
end module
```

## D.3 LNT processes

### D.3.1 Hello World program

```
module Test is
process Main [Output: any] is
   Output ("Hello World!")
end process
end module
```

### D.3.2 Pattern matching in a rendezvous

In contrast to LOTOS, offers in an LNT rendezvous can use pattern matching. Consider the type `T` and channel `C`, defined as follows:

```
type T is
   Request (x: Nat),
   Response (y: Bool)
end type

channel C is (T) end channel
```

The following process repeatedly accepts rendezvous on gate `G` (of type `C`) if the offer is a request with a value equal to either 3 or 4:

```
process P [G: C] is
   var x: Nat in
      loop
         G (?Request(x) of T) where (x > 2 and x < 5)
      end loop
   end var
end process
```

### D.3.3 Array types

The following three processes illustrate the initialization, access and modification of array (see also the definition of array types in Section D.1.4).

#### Simple array

```
type Nat_Array is
  array [0 .. 2] of Nat
end type

process main [G: any] is
  var a: Nat_Array, x: Nat in
    G (?x);
    -- initialisation of all elements to x
    a := Nat_Array (x);
    G (a[0], a[1], a[2]);
```

```
    G (?x);
    -- set element 1 to the new value x
    a[1] := x;
    G (a[0], a[1], a[2]);

    ...
  end var
end process
```

## Array of records

```
type Record is
  Record (n, m: Nat)
with get, set
end type

type Record_Array is
  array [0 .. 1] of Record
end type

process main [G: any] is
  var a: Record_Array, x, y, z: Nat in
    -- initialisation of all fields to zero
    a := Record_Array (Record (0, 0));
    G (a[0].n, a[0].m, a[1].n, a[1].m);

    G (?x, ?y, ?z) where (x < 2);
    -- set element x to the record (y, z)
    a[x] := a[x].{n -> y, m -> z};
    G (a[0].n, a[0].m, a[1].n, a[1].m);

    ...
  end var
end process
```

## Two-dimensional array type

```
type Nat_Array is
  array [0 .. 1] of Nat
end type

type Nat_Array_Array is
  array [0 .. 1] of Nat_Array
end type

process main [G: any] is
  var a: Nat_Array_Array, x, y, z: Nat in
    G (?x, ?y);
    -- simultaneous initialisation of both lines
    a := Nat_Array_Array (Nat_Array (x), Nat_Array (y));
```

```
    G (a [0][0],  a [0][1],  a [1][0],  a [1][1]);

    G (?x, ?y, ?z) where (x < 2) and (y < 2);
    -- set of element (x, y) to the new value z
    var b: Nat_Array in
      b := a[x];
      b[y] := z;
      a[x] := b
    end var;
    G (a [0][0],  a [0][1],  a [1][0],  a [1][1]);

     ...
  end var
end process
```

### D.3.4 The Alternating Bit protocol

This example is a variant of the alternating bit protocol.

#### Channel definitions

The protocol uses four different kinds of channel:

- Channels connected to the environment: these channels carry a message, i.e, value of type `Msg`:

  **channel** `C` **is** (`Msg`) **end channel**

- Channels carrying pairs of a message and a bit, i.e., one value of type `Msg` and one of type `Bit`:

  **channel** `M` **is** (`Msg`, `Bit`) **end channel**

- Channels carrying a bit, i.e., a value of type `Bit`:

  **channel** `A` **is** (`Bit`) **end channel**

#### The root process `MAIN`

The complete system of the alternating bit protocol is described by the following parallel composition of four processes, encapsulated inside the root process `MAIN`.

```
process MAIN [GET, PUT: C] is
  hide SDT, RDT: M, RACK, SACK: A, RDTe, SACKe: none in
    par SDT, RDT, RDTe, RACK, SACK, SACKe in
      par
        TRANSMITTER [PUT, SDT, SACK, SACKe] (0 of Bit)
      ||
        RECEIVER [GET, RDT, RACK, RDTe] (0 of Bit)
      end par
    ||
      par
        MEDIUM1 [SDT, RDT, RDTe]
```

```
         ||
            MEDIUM2 [RACK, SACK, SACKe]
         end par
      end par
   end hide
end process
```

### The process TRANSMITTER

```
process TRANSMITTER [PUT: C, SDT: M, SACK: A, SACKe: none] (in var b: Bit) is
var m: Msg in
   loop
      PUT (?m);               (* receive a message *)
      loop L in
         SDT (m, b);          (* send a message *)
         alt
            SACK (b);         (* control bit correct *)
            b := not(b);
            break L
         []
            SACK (not(b))     (* control bit incorrect => resend *)
         []
            SACKe             (* indication of loss => resend *)
         []
            i                 (* timeout => resend *)
         end alt
      end loop
   end loop
end var
end process
```

### The process RECEIVER

```
process RECEIVER [GET: C, RDT: M, RACK: A, RDTe: none] (in var b: Bit) is
var m: Msg in
   loop
      alt
         RDT (?m, b);         (* control bit correct *)
         GET (m);             (* delivery of message *)
         RACK (b);            (* receipt acknowledgement send correct *)
         b := not(b)
      []
         RDT (?any Msg, not(b)); (* control bit incorrect => *)
         RACK (not(b))        (* receipt acknowledgement send incorrect *)
      []
         RDTe;                (* indication of loss => *)
         RACK (not(b))        (* receipt acknowledgement send incorrect *)
      []
         i;                   (* timeout => *)
         RACK (not(b))        (* receipt acknowledgement send incorrect *)
      end alt
```

```
   end loop
end var
end process
```

### The processes MEDIUM1 and MEDIUM2

```
process MEDIUM1 [SDT, RDT: M, RDTe: none] is
var m: Msg, b:Bit in
   loop
      SDT (?m, ?b);          (* receive a message *)
      alt
         RDT (m, b)          (* transmission correct *)
      []
         RDTe                (* loss with indication *)
      []
         i                   (* silent loss *)
      end alt
   end loop
end var
end process

process MEDIUM2 [RACK, SACK: A, SACKe: none] is
var b: Bit in
   loop
      RACK (?b);             (* receive receipt acknowledgement *)
      alt
         SACK (b)            (* transmission correct *)
      []
         SACKe               (* loss with indication *)
      []
         i                   (* silent loss *)
      end alt
   end loop
end var
end process
```

# Appendix E

# Differences between Lnt2Lotos and Traian

## E.1 Introduction

This appendix lists the differences between:

- the Lnt language defined in this reference manual, and
- the Lnt language accepted by the Traian compiler and described in [SCC+24] (or its more recent versions).

Since 2020, with the advent of Traian 3.*, a deep convergence has been undertaken to make Lnt2Lotos and Traian fully compatible. There still remain a few differences, which are listed in the present Annex.

## E.2 Keywords

The "**library**" keyword is recognized by Traian but not by Lnt2Lotos. See Section 3.3 for the list of keywords recognized by Lnt2Lotos.

## E.3 Module definitions

- Traian currently supports a "**library** ... **end library**" syntax, which is not supported by Lnt2Lotos, to split a module into several files.

## E.4 Type definitions

- The predefined library of Traian and the predefined functions available with Lnt2Lotos (See appendix C) are globally compatible. Should some Traian function not be supported by Lnt2Lotos, it is always possible to write it using Lnt, and possibly implement it as an external C function.

- Arrays, predicate types, range types, sets, and sorted lists are not yet fully implemented in Traian, which accepts them, performs the appropriate checks, but does not generate C code for them.
- The Lotos code generated by Lnt2Lotos may assume function dependencies that are not checked (neither by Lnt2Lotos itself, nor by Traian), triggering errors when compiled with Cæsar.adt and Cæsar. For instance, the "`first`" (resp. "`last`") function generated by Lnt2LOTOS for a constructed type $T$ whose fields have types $T_i$ ($i \in 1..n$) requires the existence of "`first`" (resp. "`last`") for each type $T_i$. Also, the "`pred`" (resp. "`succ`") function requires the existence of "`pred`" (resp. "`succ`"), "`first`", "`last`" and "`==`" for each type $T_i$.

## E.5 Channel definitions

- Lnt2Lotos does not yet support the definition of **raise** channels. The only "**raise**" channel is the predefined channel "**exit**".

## E.6 Function definitions

- When invoking a constructor or a function (in a pattern or an expression) or a process (in a behaviour), Lnt2Lotos supports neither parameters passed in the "named style", e.g., "$F$ `(`$X_1$ `->` $V_1$`,` ... $X_n$ `->` $V_n$`)`", nor wildcards ("**...**"), whereas Traian suppports both of them.
- Contrary to Traian, which accepts either "**any**" or "**any** $T$" in patterns, Lnt2Lotos (which performs only a limited form of type checking) only accepts "**any**" in simple contexts where the type can be inferred easily; in other contexts, it requires the use of "**any** $T$" instead.
- The "**trap**" operator is recognized by Traian, but not yet supported by Lnt2Lotos, because it cannot be implemented easily in the target language Lotos. Consequently, with Lnt2Lotos, exceptions can be raised, but not trapped. If a program raises an exception (either explicitly using the "**raise**" operator, or implicitly, e.g., because of a division by zero or selection of an improper field), the execution stops.

## E.7 Process definitions

Processes and behaviours are not yet fully implemented in Traian, which accepts them, performs the appropriate checks, but does not generate code for them.

# Appendix F

# Translation of LNT constants

This appendix provides detailed examples showing how LNT constants are translated to LOTOS by LNT2LOTOS.

The generated LOTOS code relies on the `LNT_V1` library, which provides useful auxiliary definitions. This library defines an `LntExtensions` type that contains all the types defined in the `X_ACTION`, `X_BOOLEAN`, `X_NATURAL`, `X_INTEGER`, `X_REAL`, `X_CHARACTER`, and `X_STRING` libraries and defines the minimal set of operators that support the translation into LOTOS of the LNT notations.

## F.1 Translation of LNT natural numbers to LOTOS

The following table shows how LNT natural numbers (see Section 3.5) are translated to LOTOS:

| LNT notation | LOTOS translation |
|---|---|
| `0` | `(0)` |
| `9` | `(9)` |
| `123` | `(1 DecNum 2 DecNum 3)` |
| `0x4` | `(4)` |
| `0xf` | `(Hex__F)` |
| `0xAD` | `(Hex__A HexNum Hex__D)` |
| `0xF9D8` | `(Hex__F HexNum 9 HexNum Hex__D HexNum 8)` |
| `0o5` | `(5)` |
| `0o76` | `(7 OctNum 6)` |
| `0o746` | `(7 OctNum 4 OctNum 6)` |
| `0b1` | `(1)` |
| `0b1011` | `(1 BinNum 0 BinNum 1 BinNum 1)` |
| `0b110` | `(1 BinNum 1 BinNum 0)` |

The translation to LOTOS of the natural numbers is easily readable. Since infix operators are left-associative, we get the following expression:

```
(((1 DecNum 2) DecNum 3) DecNum 4)
```

from the `1234` number that we could also have manually written:

```
1 DecNum 2 DecNum 3 DecNum 4
```

One must be careful not to write strange LOTOS numbers such as `3 BinNum 2` or `f DecNum 8`. They will be interpreted as `3*2+2` and `15*10+8` by the CADP tools, but they are not valid notations of binary and decimal numbers.

In the context of hexadecimal numbers ("`0x`..." strings) the digits `a` to `f` are respectively translated to the constants `Hex__A` to `Hex__F`.

## F.2 Translation of LNT integer numbers to LOTOS

The following table shows how LNT integer numbers (see Section 3.6) are translated to LOTOS:

| LNT notation | LOTOS translation |
|---|---|
| 0 | (0) |
| 000 | (0) |
| 12 | (1 DecNum 2) |
| 123 | (1 DecNum 2 DecNum 3) |
| 0123 | (1 DecNum 2 DecNum 3) |
| 1_2_3 | (1 DecNum 2 DecNum 3) |
| +0 | (Pos (0)) |
| +00000 | (Pos (0)) |
| +12 | (Pos (1 DecNum 2)) |
| +123 | (Pos (1 DecNum 2 DecNum 3)) |
| +1_2_3 | (Pos (1 DecNum 2 DecNum 3)) |
| +0123 | (Pos (1 DecNum 2 DecNum 3)) |
| -0 | (Pos (0)) |
| -000 | (Pos (0)) |
| -1 | (Neg (0)) |
| -9 | (Neg (8)) |
| -12 | (Neg (0) DecNum -(2)) |
| -00012 | (Neg (0) DecNum -(2)) |
| -123 | (Neg (0) DecNum -(2) DecNum -(3)) |
| 0x4 | (4) |
| 0x4f | (4 HexNum Hex__F) |
| 0xab | (Hex__A HexNum Hex__B) |
| 0xa_b | (Hex__A HexNum Hex__B) |
| +0xab | (Pos (HexNum__A HexNum Hex__B)) |
| +0xa_b | (Pos (HexNum__A HexNum Hex__B)) |
| -0xa | (Neg (9)) |
| -0xb | (Neg (Hex__A)) |
| -0x0003 | (Neg (2)) |
| -0xFD | (Neg (e) HexNum -(Hex__D)) |
| -0x000789a | (Neg (6) HexNum -(8) HexNum -(9) HexNum -(Hex__A)) |

| LNT notation | LOTOS translation |
|---|---|
| `0o12` | `(1 OctNum 2)` |
| `0o1_2` | `(1 OctNum 2)` |
| `+0o12` | `(Pos (1 OctNum 2))` |
| `+0o1_2` | `(Pos (1 OctNum 2))` |
| `-0o76` | `(Neg (6) OctNum -(6))` |
| `-0o002` | `(Neg (1))` |
| `-0o00234` | `(Neg (1) OctNum -(3) OctNum -(4))` |
| `0b11` | `(1 BinNum 1)` |
| `0b1_1` | `(1 BinNum 1)` |
| `+0b11` | `(Pos (1 BinNum 1))` |
| `+0b1_1` | `(Pos (1 BinNum 1))` |
| `-0b001` | `(Neg (0))` |
| `-0b00101` | `(Neg (0) BinNum -(0) BinNum -(1))` |
| `-0b1011` | `(Neg (0) BinNum -(0) BinNum -(1) BinNum -(1))` |

The translation to LOTOS adds surrounding parentheses to all numerical constants and removes leading zeros (following the prefix indicating the base, if any). To avoid overflows[1], a negative constant (i.e., a number preceded by a unary minus operator "`-`") is translated using the constructor "`Neg()`" for the first digit. Notice that because "`Neg(X)`" is defined as "`-X-1`", it is necessary to decrement the first digit and to treat "`0`" as a special case. The unary plus operator "`+`" is translated by the constructor "`Pos()`".

Note that the use of the use of explicit prefixes "`+`" and "`-`" generates expressions using the constructors "`Pos()`" and "`Neg()`", avoiding the need for explicit type annotations (e.g., "` of Int`").

| Input LNT sequence | LOTOS translation |
|---|---|
| `n - 1` | `n - (1)` |
| `n-1` | `n-(1)` |
| `m - 1 + n` | `m - (1) + n` |
| `n == -1` | `n == -(1)` |
| `return-1` | `return-(1)` |
| `\| -1 ->` | `\| -(1) ->` |

When natural numbers and integer numbers need to be used in the same specification, number notations have to be explicitly cast:

- `12 of Nat` will be translated to `(1 DecNum 2) of Nat`
- `12 of Int` will be translated to `(1 DecNum 2) of Int`

## F.3 Translation of LNT real numbers to LOTOS

This section explains how LNT floating-point numbers (see Section 3.7) are translated to LOTOS.

Floating-point numbers are translated into a call to the LOTOS operation "`Real()`" that takes a character string as argument and is implemented by a call to "`strtod()`".

[1] Using $n$ bits, the constant $2^{n-1}$ cannot be represented.

| LNT notation | LOTOS translation |
|---|---|
| 0.1 | Real (Real__0 ~ Real__Dot ~ Real__1) |
| 0.2 | Real (Real__0 ~ Real__Dot ~ Real__2) |
| 3.0e-1 | Real (Real__3 ~ Real_Dot ~ Real__0 ~ Real_Exp ~ Real_Neg ~ Real__1) |
| 4.e0 | Real (Real__4 ~ Real_Dot ~ Real_Exp ~ Real__0) |
| 5.0 | Real (Real__5 ~ Real_Dot ~ Real__0) |

## F.4 Translation of LNT characters to LOTOS

This section explains how LNT characters (see Section 3.8) are translated to LOTOS.

Each character is translated into `Char__`*iii*, where *iii* is the decimal ASCII code of the character written with 3 digits ($iii \leq 255$).

The character constants can also be written using these operators. The following example shows the translation into LOTOS of some character constants:

| LNT notation | LOTOS translation |
|---|---|
| 'Z' | Char__090 |
| '0' | Char__048 |
| '\0' | Char__000 |
| '\n' | Char__010 |
| '\'' | Char__039 |
| '\"' | Char__034 |
| '\x5A' | Char__090 |
| '\132' | Char__090 |

## F.5 Translation of LNT strings to LOTOS

This section explains how LNT strings (see Section 3.9) are translated to LOTOS.

Each string literal constant is translated into the concatenation of predefined strings made of one character only. The concatenation operator ~ is an internal one that must be used only to concatenate string literal constants. All string literal constants of one character are implemented by operators `String__iii` where `iii` is the decimal ASCII code of the character written with 3 digits (code less than or equal to 255). Each string ends with `String__000`.

The string constants can also be written using these operators. The translation into LOTOS of the strings of the previous example is:

```
String (String__000 (* "" *))
String (String__233 ~ String__234 ~ String__232 ~ String__000 (* "éêè" *))
String (String__050 ~ String__010 ~ String__108 ~ String__105 ~ String__110
        ~ String__101 ~ String__115 ~ String__000 (* "2\nlines" *))  ;
String (String__034 ~ String__000 (* "\"" *))  ;
String (String__039 ~ String__000 (* "'" *))  ;
String (String__039 ~ String__000 (* "\'" *))  ;
String (String__092 ~ String__000 (* "\\" *))  ;
String (String__065 ~ String__090 ~ String__069 ~ String__082 ~ String__084
        ~ String__089 ~ String__000 (* "AZERTY" *))  ;
String (String__065 ~ String__090 ~ String__069 ~ String__082 ~ String__084
        ~ String__089 ~ String__000 (* "A\x5AERTY" *))  ;
```

```
String (String__065 ~ String__090 ~ String__069 ~ String__082 ~ String__084
        ~ String__089 ~ String__000 (* "A\132ERTY" *))  ;
```

This translation uses the fact that the C pre-compilers support string literal constants constructed from contiguous shorter strings separated by simple spaces:

```
printf ("H" "e" "ll" "o") ;
```

The `"ABC"` String constant that is translated to the expression `(String__065 ~ ... ~ String__000)` is then compiled to LOTOS with the following result:

```
... = (((STRING__065 ~ STRING__066) ~ STRING__067) ~ STRING__000) OF STRING;
```

This equation is then compiled by CÆSAR and CÆSAR.ADT to generate the following C code:

```
return ADT_CONCAT_CONST_STRING (
         ADT_CONCAT_CONST_STRING (
           ADT_CONCAT_CONST_STRING (ADT_STRING_065 (),
                                    ADT_STRING_066 ()),
           ADT_STRING_067 ()),
         ADT_STRING_000 ());
```

The C macro definitions of `ADT_CONCAT_CONST_STRING` and `ADT_STRING_`*iii* finally generate:

```
return "A"  "B"  "C"  "\x00";
```

which is equal to:

```
return "ABC\0";
```

# Appendix G

# Change history

In May 2014, the contents of this appendix have been moved to the `$CADP/HISTORY` file, as a logical consequence of the fact that since January 2010, Lnt2Lotos and related tools are integral part of the Cadp toobox.

The following table gives the mapping between the versions of Lnt2Lotos and the corresponding items in the `$CADP/HISTORY` file.

| Version | Release date | `$CADP/HISTORY` item(s) |
|---|---|---|
| 1A | Jul. 8, 2005 | #1457 part 1 |
| 1B | Sep. 16, 2005 | #1457 part 2 |
| 1C | Sep. 29, 2005 | #1457 part 3 |
| 1D | Oct. 20, 2005 | #1457 part 4 |
| 1E | Feb. 24, 2006 | #1457 part 5 |
| 1F | Mar. 15, 2006 | #1457 part 6 |
| 2A | Apr. 21, 2006 | #1457 part 7 |
| 2D | Jun. 1, 2006 | #1457 part 8 |
| 2F | Feb. 5, 2007 | #1457 part 9 |
| 2G | Jun. 5, 2007 | #1457 part 10 |
| 3A | Jun. 8, 2007 | #1457 part 11 |
| 3B | Apr. 7, 2008 | #1457 part 12 |
| 3C | May 19, 2008 | #1457 part 13 |
| 4A | Jun. 19, 2008 | #1457 part 14 |
| 4B | Jul. 18, 2008 | #1457 part 15 |
| 4C | Aug. 1st, 2008 | #1457 part 16 |
| 4D | Sep. 8, 2008 | #1457 part 17 |
| 4E | Sep. 11, 2008 | #1457 part 18 |
| 4F | Oct. 15, 2008 | #1457 part 19 |
| 4G | Jan. 14, 2009 | #1457 part 20 |
| 4H | Apr. 10, 2009 | #1457 part 21 |
| 4I | Jun. 24 2009 | #1457 part 22 |
| 4J | Sep. 28, 2009 | #1457 part 23 |
| 4K | Dec. 4, 2009 | #1457 part 24 |

| Ver. | Release date | `$CADP/HISTORY` item(s) |
|---|---|---|
| 5.0 | Sep. 14, 2010 | #1571 part 1 |
| 5.1 | Feb. 13, 2011 | #1571 part 2 |
| 5.2 | May 17, 2011 | #1571 part 3 |
| 5.3 | Jul. 5, 2011 | #1571 part 4 |
| 5.4 | Sep. 12, 2011 | #1571 part 5 |
| 5.5 | Feb. 20, 2012 | #1591 |
| 5.6 | Jul. 5, 2012 | #1594, #1609, #1610, #1619, #1620, #1622, #1623 |
| 5.7 | Jan. 11, 2013 | #1640, #1641, #1642, #1643, #1645, #1646, #1647, #1648, #1649, #1650, #1653, #1654, #1655 |
| 5.8 | Dec. 13, 2013 | #1661, #1662, #1663, #1667, #1668, #1669, #1670, #1677, #1681, #1683, #1685, #1686, #1687, #1688, #1689, #1697, #1698, #1699, #1700, #1707, #1708, #1723, #1725, #1734, #1739, #1741, #1750, #1751, #1752, #1760 |
| 5.9 | Jan. 13, 2014 | #1766, #1767, #1770, #1771 |
| 6.0 | May 13, 2014 | #1776, #1777, #1778, #1784, #1786, #1787, #1788, #1790, #1792, #1794, #1796, #1798, #1799, #1800, #1805, #1810, #1811, #1813, #1824, #1825, #1826, #1830, #1831, #1834, #1836, #1837, #1838, #1839, #1843, #1845, #1846, #1847, #1850, #1851, #1852, #1853, #1854, #1856, #1857, #1858, #1859, #1861, #1862, #1863, #1865, #1872, #1875, #1878, #1881 |
| 6.1 | Aug. 26, 2014 | #1887, #1939, #2007 |
| 6.2 | Feb. 26, 2015 | #2018, #2024, #2026, #2032, #2034, #2036, #2043, #2045, #2047, #2049, #2053, #2060, #2064 |
| 6.3 | Jul. 26, 2015 | #2075, #2076, #2087, #2088, #2098, #2100, #2103, #2109, #2110, #2111, #2112, #2113, #2117, #2119, #2122, #2124, #2126, #2129, #2131, #2132, #2134, #2138, #2139 |
| 6.4 | Jan. 13, 2016 | #2141, #2147, #2149, #2150, #2152, #2154, #2156, #2166, #2170, #2173, #2201, #2217, #2218, #2219, #2221, #2225, #2228, #2230, #2234, #2235 |
| 6.5 | Sep. 13, 2016 | #2239, #2241, #2242, #2245, #2251, #2269, #2270, #2271, #2275, #2276, #2278, #2279, #2281, #2283 |
| 6.6 | Mar. 26, 2017 | #2286, #2288, #2289 |
| 6.7 | Jul. 13, 2017 | #2291, #2292, #2295, #2300, #2310, #2314 |
| 6.8 | Jan. 13, 2019 | #2317, #2319, #2321, #2322, #2323, #2325, #2327, #2332, #2342, #2343, #2346, #2347, #2350, #2352, #2354, #2361, #2390, #2450, #2462, #2478 |
| 6.9 | | to be completed |

# Bibliography


[Gar95] Hubert Garavel. On the Introduction of Gate Typing in E-LOTOS. In Piotr Dembinski and Marek Sredniawa, editors, *Proceedings of the 15th IFIP International Workshop on Protocol Specification, Testing and Verification (PSTV'95), Warsaw, Poland*, pages 283–298. Chapman & Hall, June 1995.

[Gar15] Hubert Garavel. Revisiting Sequential Composition in Process Calculi. *Journal of Logical and Algebraic Methods in Programming*, 84(6):742–762, November 2015.

[GLS17] Hubert Garavel, Frédéric Lang, and Wendelin Serwe. From LOTOS to LNT. In Joost-Pieter Katoen, Rom Langerak, and Arend Rensink, editors, *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, volume 10500 of *Lecture Notes in Computer Science*, pages 3–26. Springer, October 2017.

[GS96] Hubert Garavel and Mihaela Sighireanu. On the Introduction of Exceptions in LOTOS. In Reinhard Gotzhein and Jan Bredereke, editors, *Proceedings of the IFIP Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'96), Kaiserslautern, Germany*, pages 469–484. Chapman & Hall, October 1996.

[GS99] Hubert Garavel and Mihaela Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In Jianping Wu, Qiang Gao, and Samuel T. Chanson, editors, *Proceedings of the IFIP Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'99), Beijing, China*, pages 185–202. Kluwer Academic Publishers, October 1999.

[ISO89] ISO/IEC. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva, September 1989.

[ISO01] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization – Information Technology, Geneva, September 2001.

[SCC+24] Mihaela Sighireanu, Alban Catry, David Champelovier, Hubert Garavel, Frédéric Lang, Guillaume Schaeffer, Wendelin Serwe, and Jan Stoecker. LOTOS NT User's Manual (Version 3.14). INRIA/CONVECS, Grenoble, France, `https://vasy.inria.fr/ftp/traian/manual.pdf`, 88 pages, June 2024.