LOTOS NT User Manual[*]

Mihaela Sighireanu

with updates by Alban Catry, David Champelovier, Hubert Garavel,
Frédéric Lang, Guillaume Schaeffer, Wendelin Serwe, and Jan Stöcker

Release 3.8 — October 29, 2022

(*) There are two slightly different versions of the Lotos NT language: one is supported by the TRAIAN compiler, the other (now called LNT for short) is supported by the LNT2LOTOS, LNT.OPEN, and LPP tools of the CADP toolbox. The present manual describes the TRAIAN version, which is mostly used for compiler construction. For additional information about Lotos NT and LNT, see `http://cadp.inria.fr/tutorial`.

Foreword

The present User Manual of Lotos NT comes with the release 3.8 of the compiler Traian. It completely describes the syntax and the informal semantics of the subset of Lotos NT currently supported by Traian.

Availability The complete distribution for Traian is available on the web at the address `http://vasy.inria.fr/traian`. Please report feedback and bugs to `cadp@inria.fr`.

Acknowledgments The author thanks people who helped in writing this manual, either through their ideas or their comments about the different versions of this manual. I owe thanks to Frédéric Lang, and also to Fabrice Baray, Claude Chaudet, Hubert Garavel, Marc Herbert, Radu Mateescu, and Bruno Vivien.

Contents

1 Introduction **9**
1.1 Background . . . 9
1.2 Goals . . . 11
1.3 Main Concepts . . . 11
1.4 LOTOS NT versus E-LOTOS . . . 12
1.5 Manual Structure . . . 13

2 Basic mathematical concepts and notation **15**
2.1 General . . . 15
2.2 Backus-Naur Form . . . 16
2.3 Description of the Syntax . . . 17
2.4 Data values . . . 17

3 Lexical Structure **19**
3.1 ISO Latin-1 Character Set . . . 19
3.2 Input Elements and Tokens . . . 20
3.3 Comments . . . 20
3.4 Includes . . . 20
3.5 Identifiers . . . 21
3.6 Special Identifiers . . . 21
3.7 Keywords . . . 22
3.8 Literals . . . 22
3.8.1 Integer Literals . . . 23
3.8.2 Floating-Point Literals . . . 24
3.8.3 Characters . . . 24
3.8.4 String Literals . . . 25
3.9 Operators . . . 26

4 Modules **27**
4.1 Module Definition . . . 27
4.2 Module Pragmas . . . 27

5 Types **29**
5.1 Type Definition . . . 29
5.2 Predefined Operations . . . 30
5.3 Predefined Types . . . 31
5.3.1 The boolean type . . . 32
5.3.2 The natural type . . . 32

5.3.3 The integral type 32
5.3.4 The floating point type 33
5.3.5 The character type 33
5.3.6 The string type 34
5.4 Derived Types 34
5.4.1 Enumerated types 35
5.4.2 Scalar and simple types 35
5.4.3 Record types 35
5.4.4 Lists 36
5.5 Sorted lists 36
5.6 Sets 37
5.7 Sorted sets 37
5.8 Arrays 37
5.9 Ranges 37
5.10 Predicate types 38
5.11 External Types and Pragmas 38

6 Expressions, Statements, and Functions 41
6.1 Constants 41
6.2 Value expressions 42
6.2.1 Variables 44
6.2.2 Constructor application 44
6.2.3 Function application 45
6.2.4 Field selection 45
6.2.5 Field update 46
6.2.6 Unambiguous Expressions 46
6.3 Patterns 47
6.4 Statements 49
6.4.1 Value return 51
6.4.2 Null Statement 51
6.4.3 Assignment 51
6.4.4 Sequential Composition 51
6.4.5 Variable declaration 51
6.4.6 Case statement 52
6.4.7 If statement 53
6.4.8 Iteration Statements 54
6.4.9 Events and their handling 56
6.4.10 Variable use 58
6.4.11 Event access 58
6.5 Functions 58
6.5.1 Function definition 59
6.5.2 Function call 61

7 Channels, Behaviours, and Processes 63
7.1 Channels 63
7.2 Behaviours 63
7.2.1 Stop Behaviour 65
7.2.2 Rendezvous 65
7.2.3 Sequential Composition 65
7.2.4 Process Call 66

7.2.5 Function call 66
7.3 Process Definition 67

A Syntax Summary 69
A.1 Syntax of the module part 69
A.2 Syntax of the data part 71
A.3 Syntax of the behaviour part 75

Bibliography 79

Index 81

Chapter 1

Introduction

> "*Formal description techniques (FDT) are methods of defining the behaviour of an (information processing) system in a language with formal syntax and semantics, instead of a natural language as English.*" [ISO-8807]

In the following sections, the origin and the evolution of FDTs are discussed, especially LOTOS. The objectives that the new generation of FDTs must satisfy are considered. The main concepts of LOTOS NT and E-LOTOS are presented. Finally the structure of the document is explained.

Note: The present chapter was written in 2000, and slightly updated afterwards. For a more recent introduction and comparison of LOTOS, LOTOS NT, and E-LOTOS, see also [GLS17].

1.1 Background

In the 80s, three formal description techniques (ESTELLE, LOTOS, and SDL) have been standardized at the international level to precisely describe (better than using natural language, which is always ambiguous) the services and protocols used in telecommunication and networked computer systems.

LOTOS was defined within ISO during the years 1981–1989. The objectives of its design follow strictly the main general objectives defined for FDTs:

- *expressive*: LOTOS was found capable of describing both the protocols and services of the seven layers of OSI=[1] reference model.
- *well-defined*: LOTOS has a formal mathematical model suitable for the analysis of descriptions supported by the testing of an implementation for conformance.
- *well-structured*: LOTOS offers many means for structuring of specification.
- *abstract*: LOTOS is independent from the methods of implementations and offers means for abstraction of irrelevant details.

As a design choice, LOTOS consists of two "orthogonal" sub-languages:

[1] Open System Interconnection

The data part of LOTOS is dedicated to the description of data structures. It is based on the well-known theory of algebraic abstract data types [Gut77], more specifically on the ACTONE specification language [dMRV92].

The control part of LOTOS is based on the process algebra approach for concurrency, and appears to combine the best features of CCS [Mil89] and CSP [Hoa85].

LOTOS has been applied to describe complex systems formally, for example: the service and protocols for the OSI transport and session layers [ISO89b, ISO89a, ISO92b, ISO92c], the CCR[2] service and protocol [ISO95b, ISO95a], OSI TP[3] [ISO92a, Annex H], MAA[4] [Mun91], FTAM[5] basic file protocol [LL95], etc. It has been mostly used to describe software systems, although there are recent attempts to use it for asynchronous hardware descriptions [CGM+96].

A number of tools have been developed for LOTOS, covering user needs in the areas of simulation, compilation, test generation, and formal verification.

Nevertheless, the three FDTs, including LOTOS, actually show their limitation for several reasons:

- Some design choices must be revised in order to respond to criticism of users. For example, the abstract data types used in LOTOS and SDL do not satisfy users.
- The new communication protocols like those of high flow network (e.g., ATM) or multimedia protocols need the specification of real-time constraints. None of the three FDTs allows to express all needed quantitative temporal constraints.
- The development of new architectures like ODP[6] or CORBA[7] call into question the OSI reference model and its static architecture. The model chosen is more dynamic and mobility is important.

For these reasons ISO/IEC undertook a revision of the LOTOS standard in 1993. The revised language is called E-LOTOS (for Extended-LOTOS). The enhancements of LOTOS should remove known limitations of the language concerning expressiveness, abstraction and structuring capabilities, user friendliness. A non-exhaustive list of such undesirable characteristics is given below:

- Despite having a strong mathematical basis, the abstract data types need a good background from the part of users. This prevents the use of the language by a large public, restricting it to an "expert" public.
- LOTOS is able to describe only temporal ordering, for example "the sending of a message is followed by its reception". However, one needs to express quantitative time requirements like "the sending of a message is followed after 5 seconds by the message reception".
- In the control part, the value passing is done in a pure functional style. Despite its proper semantics, this feature adds cumbersome constraints for structuring the specification. For this reason, a lot of case studies are done using "Basic LOTOS", *i.e.*, LOTOS without values.

[2] Commitment, Concurrency, and Recovery
[3] Distributed Transaction Processing
[4] Message Authentication Algorithm
[5] File Transfer, Access, and Management
[6] Open Distributed Processing
[7] Common Object Representation Brooker Architecture

1.2 Goals

This section lists a number of qualities which, in our opinion, the new generation of FDT languages should have.

The first of them is that E-Lotos must be a useful and pleasant *tool* for behaviour description and analysis, and not a set of more or less awkward constraints.

This language must be *easy to learn*, which implies that its constructions have a *well defined, non-ambiguous semantics*, and that it respects (in the limits of the semantics) most rules and habits of the users. Because programming languages are intuitive and their concepts largely used, providing the language with algorithmic features is a mean to accomplish this goal.

The language should also provide as much as possible *description safety and reliability*, while remaining very *versatile*. As many errors as possible must be detected at compile time.

The language should provide maximal *expressiveness*. For example, the use of Lotos in several case studies showed that the operators and the concepts of the language are not able to express self interruption of behaviour [QA92], deterministic control passing between processes [GH93], or nets of processes [Bol90]. E-Lotos should fill these gaps. Expressiveness also concerns the means for describing real-time aspects. Actually, there exists several extensions of Lotos with real-time operators: ET-Lotos [LL97], RT-Lotos [Cd95]. These extensions form a strong basis for the definition of real-time aspects of E-Lotos. As an extension of Lotos, the language should provide mean for upward compatibility: a translation of Lotos constructs in E-Lotos should be provided.

The language should allow the *modular* description of systems and description *re-usability*.

In the context of the Odp group at Iso/Iec, the language should provide means for an *easy interface* with external description or programming languages developed by this group, e.g., Idl. Also, it should remain independent from, but *easily translatable* into most implementation languages (first targets being C, Ada, Java). The accomplishment of this goal will provide a good platform for tools developers, and so a possible large distribution of the language.

The language should provide constructs offering opportunities for an *optimal analysis* by tools.

Last but not least, the language must be *simple*.

1.3 Main Concepts

This section presents the main concepts of E-Lotos, together with a short justification of their introduction in the language. Those justifications are related to the goal listed in the previous section.

First of all, the E-Lotos main feature is *concurrency*. It is a mean for description of concurrent (parallel) evolution of systems and their communication. The systems are composed in parallel using a Csp-like [Hoa78] operator. The base mechanism for communication is the *rendezvous* on communication points called *gates*. The communication allows the exchange of values. This is the only mean for interaction between concurrent systems because their memory spaces must be disjoint. The language provides several mechanisms for concurrency: interleaving, binary and n-ary synchronization, network synchronization, coroutine mechanism. This is a first step towards language expressiveness.

E-Lotos is a description language supporting non-determinism. Both internal and external [Hoa78] non-determinism is provided. By difference with Lotos, E-Lotos provides also deterministic choice constructs by means of “if-then-else” and “case” statements. The introduction of these constructs touches both easy to use and optimal analysis requirement.

The language provides means for *real-time* descriptions. All operators of the language have an intuitive time semantics. The time domain may be defined by the user with respect to a proper semantics. So the time domain may be dense or discrete.

In the same frame of expressiveness, E-Lotos supports *exception handling* in order to deal with abnormal conditions. The exceptions are modeled by signals.

The language is *strongly typed*, a necessary condition for description safety. All objects in a description must be typed. *Type checking* is performed on any E-Lotos description in order to detect, at compile time, most inconsistencies and errors. Basic types include integers, reals, booleans, strings, etc. User defined types may be defined by using type constructors. This provides means for defining most usual types: enumeration types, records, unions, sets, lists. Types may be recursive. Also, types and functions may be specified into an external language.

E-Lotos remains a *functional* language in its semantics, although it supports assignment of variables. This is a step toward user friendliness on the one hand, and interfacing with external languages, on the other hand.

Modularity is a basic feature of E-Lotos. Constants, types, functions, and processes may be defined in separate *modules*. The modules support the definition of local objects (constants, types, functions, and processes). The visibility of local objects is specified by means of module *interfaces*. Modules may be combined by *importation*. Another important feature for re-usability purposes is *genericity*. Generic modules provide means for parameterizing modules with constants, types, functions, and processes. As in Lotos, the dynamic semantics of behaviours and expressions are given only for fully instantiated modules.

1.4 LOTOS NT versus E-LOTOS

Lotos NT is the language supported by the Traian compiler. It follows the main concepts of E-Lotos and offers other features, in order to provide versatility, compilation and verification efficiency.

[Sig99] exposes the main differences between Lotos NT and E-Lotos. We cite only two examples:

- In Lotos NT, function names may be *overloaded* as in Ada [WWF87], *i.e.*, two or more functions may have the same name provided they have different *profiles* (list of parameter types and result type). This is a useful feature because it improves the semantic consistency of a Lotos NT specification—two similar operations on different types need not have different names—and the semantic consistency of Lotos NT predefined functions themselves. Also, the compatibility with ActOne is ensured.

- In Lotos NT, functions may have input, output, and input/output parameters as in Ada. This provides means for returning several results and for easy interfacing with languages as Idl.

- The style of the Lotos NT language is fully imperative in syntax and semantics, unlike E-Lotos which has functional semantics.

1.5 Manual Structure

This manual gives an informal definition of the Lotos NT language. A formal definition may be found in [Sig99].

Chapter 2 presents the mathematical notations and concepts used. Chapter 3 presents the lexical structure of the language. Chapter 4 presents the modules. The next language constructs are presented bottom-up, in order to make the language easier to learn. We begin by presenting types and type declarations in chapter 5. The language of data is presented in chapter 6. It contains data expressions, statements, and function declarations.

Each section of the chapters defining the language presents language constructs in the following order:

- the goals and the rationales of the construct;
- its abstract syntax;
- its intuitive and its formal, static, and dynamic semantics;
- some examples of its use.

Annex A presents the full syntax of the language.

We tried to present the information in a strictly linear order. However, where it is not possible to do so, we signal forward references.

Chapter 2

Basic mathematical concepts and notation

2.1 General

This section contains a list of basic mathematical concepts and related notations used in the remainder of the document.

$\overset{def}{=}$	is defined as.
iff	if and only if, *i.e.*, double implication.
$\{a, b, c, ...\}$	the *set* made up of elements $a, b, c, ...$. The order in which the elements are listed is immaterial.
$\emptyset$	the *empty set*.
$x \in A$	x is an *element* of the set A.
$x \notin A$	x is *not* an *element* of the set A.
$A \subseteq B$	A is a *subset* of B.
$A \times B$	the *Cartesian product* of A with B, *i.e.*, the set of all ordered pairs $< a, b >$ such that $a \in A$ and $b \in B$.
$A_1 \times A_2 \times ... \times A_n$	the *generalized Cartesian product* of $A_1, A_2, ..., A_n$, *i.e.*, the set of ordered tuples $< a_1, a_2, ..., a_n >$, such that $(\forall i) a_i \in A_i$.
$\{x \in A \mid Q(x)\}$	the set which contains only all those elements of A which satisfy the property Q.
$a_1, ..., a_n$	the finite (or empty) list (or sequence, or n-tuple) made up of the *elements*, or *components* $a_1, ..., a_n$. Unlike sets, lists may contain more than one instance of the same element, since elements are distinguished by their position in the ordering of the list; the length of the list is n;
$<>$	the *empty* list has no elements, its length is 0;
$a_0, ..., a_n$	the non-empty finite list made up of the elements $a_0, ..., a_n$; the length of the list is $n + 1$;

$\overline{a}$ — the non-empty finite list made up of the elements $a_0, ..., a_n$;
the length of the list is $len(\overline{a})$;
a *record* is a n-tuple of which each element is *labelled* with a unique label. If *lab* is the label of element x of record y, then $y.lab$ denotes x.

$R \subseteq A \times B$ — R is a *binary relation* between A and B, *i.e.*, a set of elements of $A \times B$;
the *domain* of R is defined as $\{a \in A \mid \exists b \in B. < a, b > \in R\}$;
the *range* of R is defined as $\{b \in B \mid \exists a \in A. < a, b > \in R\}$;

$\{\}$ — the empty relation;

$f : A \rightarrow B$ — f is a (*partial*) *function* (*finite map*) from A to B, *i.e.*, f is a binary relation between A and B such that for each $a \in A$ there exists at most one $b \in B$ such that $< a, b > \in f$;
the domain of f is denoted by $\mathrm{Dom}(f)$;
the range of f is denoted by $\mathrm{Ran}(f)$;
if $< a, b > \in f$ then f is *defined* for a, also written $f(a) = b$ or $a \mapsto b$;
the function f is total iff it is defined for all $a \in A$;
a function $f : A \rightarrow B$ is *injective* iff, for all a_1, a_2 in the domain of f, $f(a_1) = f(a_2)$ implies that $a_1 = a_2$;

$f : A_1 \times A_2 \times ... \times A_n \rightarrow B$ — the function from the Cartesian product $A_1 \times A_2 \times ... \times A_n$ to B;
the function *arity* maps f to the number n of terms of the Cartesian product.

2.2 Backus-Naur Form

The meta-language used in this manual to specify the syntax is based on Backus-Naum Form (BNF). A BNF description of a language L is given by a set of *productions*, or re-write rules. The meta-symbols used to compose rewrite rules are listed in Table 2.1.

Meta-symbol	Name	Pronunciation
`"xyz"`	terminal symbol	xyz
`abc`	nonterminal symbol `abc`	(nonterminal) abc
`::=`	rewrite symbol	is defined to be
`|`	alternation symbol	or, alternatively
`[...]`	option operator	0 or 1 instances of
`{...}`	repetition operator	0 or more instances of
`;`	semi-colon	end of BNF rule

Table 2.1: Meta-language symbols

A *terminal symbol* is a symbol that appears literally in L. A *nonterminal symbol* is a symbol that denotes a syntactic construct of L (which is ultimately represented by a string of terminal symbols).

A rewrite rule has the form:

`<nonterminal-symbol> ::= meta-expression ;`

where the meta-expression is an expression constructed using terminal and nonterminal symbols, and the operators listed in Table 2.1 except `::=` and `;`. Adjacent terminal or/and nonterminal symbols occurring in a meta-expression denote the lexical concatenation of the texts they ultimately represent. Concatenation respects the rules given in 3.

A rewrite rule is interpreted as follows: the nonterminal symbol of the left-hand side can be replaced by any one of the of the sequences separated by the alternation symbol.

All operators (including implicit concatenation) have precedence order over the alternative operator.

2.3 Description of the Syntax

Descriptions of concrete syntax give formal rules to be implemented by a parser for the language. Concrete syntax descriptions obey to constraints dictated by the implementation on a computer.

However, the purpose of this document is to present the syntax to the user of the language. In order to be more easily readable, we can abstract out some implementation details, and provide a more informal presentation of the concrete syntax, using meta level syntactic facilities. A human will understand the description better and faster than if written in a language designed for a machine. It uses type-setting conventions which facilitate the user reading. The conventions used for the presentation of the syntax are the following:

- terminals are represented using bold face;
- the special symbols are represented using teletype font. Note the difference between the special symbols “`[`” and “`]`” and the (mathematical style) symbols “[” and “]” used to express optional syntactic clauses in BNF.
- a non-empty list is represented like “$a_0\,,\ldots,a_n$”, *i.e.*, with the indexes starting at 0. The possibly empty lists are indexed from 1, *i.e.*, “$a_1\,,\ldots,a_n$”.

  More precisely, the BNF equivalent of “$a_0\,,\ldots,a_n$” is “$a\{\,,a\}$”, while the BNF equivalent of “$a_1\,,\ldots,a_n$” is “$[a\{\,,a\}]$”.

2.4 Data values

A *data domain* D is a set of sets; the elements of D are referred to as data carriers.

Chapter 3

Lexical Structure

This chapter presents[1] the lexical conventions of LOTOS NT.

LOTOS NT programs uses the ISO Latin-1 (8859.1) character set.

The characters resulting from the lexical translations are reduced to a sequence of input elements (§ 3.2, p. 20), which are spaces, comments (§ 3.3, p. 20), and tokens. The tokens are: identifiers (§ 3.5, p. 21), keywords (§ 3.7, p. 22), literals (§ 3.8, p. 22), and operators (§ 3.9, p. 26) of the LOTOS NT syntactic grammar.

3.1 ISO Latin-1 Character Set

The ISO Latin-1 character set is divided into:

- alphabetic characters (letters), made of ASCII [2] characters (octal codes #101–#132) and other ISO Latin 1 characters (octal codes #300–#377). See Table 2 of ISO/IEC DIS 14750.

```
LETTER ::= #101..#132 ;

LETTER_WITH_ACCENT ::= #300..#377 ;

ALPHABETIC_CHARACTER ::= LETTER | LETTER_WITH_ACCENT ;
```

- digits, *i.e.*, characters from “0” to “9”. See Table 3 of ISO/IEC DIS 14750.

```
DIGIT ::= "0".."9" ;
```

- spaces and formating characters, which include blanks, horizontal and vertical tabs, newlines, form feeds. See Table 5 of ISO/IEC DIS 14750.

```
SPACE  ::= HT | NL | FF | SP | LF | CR ;
```

NOTE: IDL considers also BEL, BS, but not SP.

[1] This section is an adaptation of *The ISO/IEC DIS 14750*, Section 4 (IDL Syntax and Semantics); it differs in the list of legal keywords and punctuation.

[2] ASCII (ANSI X3.4) is the American Standard Code for Information Interchange.

Except for comments, identifiers, and the contents of character and string literals, all input elements in a LOTOS NT specification are formed only from ASCII characters (or escapes which result in ASCII characters).

3.2 Input Elements and Tokens

The input characters and line terminators are reduced to a sequence of input elements. Input elements which are not blank spaces or comments are tokens. Tokens are the terminal symbols of the LOTOS NT syntactic grammar.

```
Input ::= [ InputElement { InputElement } ] ;

InputElement ::= SPACE | Comments | Token ;

Token ::= IDENTIFIER | Keyword | Literal | Operator | Separator ;
```

There are four classes of tokens: identifiers, keywords, literals, operators, and other separators. Blank spaces and comments are ignored except as they serve to separate tokens. Some blank space is required to separate otherwise adjacent identifiers, keywords, and literals.

If the input stream has been parsed into tokens up to a given character, the next token is taken to be the longest string of characters that could possibly constitute a token.

3.3 Comments

LOTOS NT defines two kinds of comments:

- `(* text *)` A LOTOS comment; all the text from the ASCII characters `(*` to the ASCII characters `*)` is ignored.
- `-- text` A single line comment: all the text from the ASCII characters `--` to the end of the line is ignored.

Comments do not nest. The comment characters `--`, `(*`, and `*)` have no special meaning within a `--` comment or within a `(*` comment. Comments may contain alphabetic, digit, graphic, and space (but not newline) characters.

Comments are not part of the LOTOS NT description. They may be inserted anywhere between two other lexical units or left out, except when they play the role of separators.

Zero or more separators may occur between any two consecutive tokens, before the first token, or after the last token of the LOTOS NT text.

There shall be at least one token separator between any pair of consecutive tokens if the concatenation of their texts change their meaning.

3.4 Includes

The `library ... end library` sequence allows to include files in the source code. This feature is useful to write LOTOS NT descriptions in separate files. Note however that it is now superseded by module inclusion, see Chapter 4.

The include mechanism works like the `#include` in C language: a file can be included anywhere in the source code, and the lexical analyser is in charge of replacing the sequence by the content of the included file.

```
Include ::= Library { SPACE | Comments }
                """  Filename """ { SPACE | Comments }
                { "," { SPACE | Comments } Filename { SPACE | Comments } }
            End { SPACE | Comments } Library ;

Library ::= ("l" | "L") ("i" | "I") ("b" | "B") ("r" | "R")
            ("a" | "A") ("r" | "R") ("y" | "Y") ;

End ::= ("e" | "E") ("n" | "N") ("d" | "D") ;
```

`Filename` is the path to the included file. It can be either absolute or relative to the current working directory.

Several files can be included in the same `library ... end library` sequence. In this case, the files will be included in the same order as they appear in the sequence.

3.5 Identifiers

An identifier is an unlimited-length sequence of alphabetic characters, digit characters, and underscores (“_”). It must start with an alphabetic character, cannot end with an underscore, and cannot contain consecutive underscores.

```
IDENTIFIER ::=  ALPHABETIC_CHARACTER { ["_"] NORMAL_CHARACTER } ;

NORMAL_CHARACTER  ::= DIGIT | ALPHABETIC_CHARACTER ;
```

In Lotos NT, identifiers are not case-sensitive. In a given declaration scope, two identifiers that differ only in the case of their characters are considered redefinitions of one another: they will collide and yield a compilation error. When comparing two identifiers to see if they collide:

- Upper- and lower-case letters are treated as the same letter. TBL 2 in (ISO/IEC DIS 14750) defined the equivalence mapping of upper- and lower-case letters.
- The comparison does *not* take into account equivalences between diagraphs and pairs of letters (e.g., “æ” and “ae” are not considered equivalent) or equivalences between accented or not accented letters (e.g., “à” and “a” are not considered equivalent).
- All characters are significant.

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.

3.6 Special Identifiers

In order to allow a more intuitive notation for the different mathematical operators, two special classes of identifiers are introduced, namely `SPECIAL_IDENTIFIER1` and `SPECIAL_IDENTIFIER2`, built as follows:.

```
SPECIAL_CHARACTER ::= "%" | "&" | "*" | "+" | "-" | "/" | ">" | "="
                    | "<" | "@" | "\" | "^" | "~" ;

SPECIAL_IDENTIFIER1 ::= DIGIT { ["_"] NORMAL_CHARACTER } ;

SPECIAL_IDENTIFIER2 ::= SPECIAL_CHARACTER { SPECIAL_CHARACTER } ;
```

A SPECIAL_IDENTIFIER1 or SPECIAL_IDENTIFIER2 can only be used as the identifier of a constructor or function. Identifiers of modules, types, channels, variables, events, loops, and processes have to be normal identifiers satisfying the definition of IDENTIFIER given in Section 3.5.

3.7 Keywords

The symbols given in table 3.1 are keywords of Lotos NT. They are written between double quotes in the concrete syntax and in boldface in the abstract syntax.

These keywords are reserved, meaning that they cannot be used as identifiers, except "**and**", "**div**", "**mod**", "**or**", "**rem**", and "**xor**", which can be used as function identifiers.

A few of these keywords (written in italics) are not used currently, but are reserved for future use and compatibility with the LNT language supported by the CADP toolbox.

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

Table 3.1: The keywords of Lotos NT

The following symbols "comparedby", "iteratedby", "implementedby", "int_bits", "int_check", "int_inf", "int_sup", "nat_bits", "nat_check", "nat_inf", "nat_sup", "num_bits", "num_card", "printedby", "string_card", "update", and "version" are not reserved keywords; they are understood as pragma names when they follow the "!" symbol. The keyword "list" can also be used as a pragma name.

3.8 Literals

A literal is the source code representation of a value of a primitive type (§ 5.3, p. 31).

```
Literal ::= INTEGER
          | REAL
          | CHAR
          | STRING ;
```

3.8.1 Integer Literals

See Section 5.3.3 for a general discussion of the integer types and values.

Unsigned integer literals may be expressed in decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2):

```
INTEGER ::= DECIMAL_NUMBER
          | HEX_NUMBER
          | OCTAL_NUMBER
          | BINARY_NUMBER ;
```

A decimal number is either the single ASCII character 0, representing the integer zero, or consists of an ASCII digit from 1 to 9, optionally followed by one or more ASCII digits from 0 to 9, and represents a positive integer.

```
DECIMAL_NUMBER  ::= "0"
                |   NON_ZERO_DIGIT { ["_"] DIGIT } ;

NON_ZERO_DIGIT  ::=  "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

The simplest form of integer literal is simply a sequence of decimal digits. If the literal is very long, it may be convenient to split it up into groups of digits by inserting underscores ("_"), such as "123_456_789". An integer literal cannot start or end with an underscore, and cannot contain consecutive underscores. In contrast with identifiers, such underscores, are of course, of no significance other than to make the literal easier to read.

NOTE: The use of "_" character to format integers is also adopted by ADA language.

A hexadecimal number consists of the leading ASCII characters 0x followed by one or more ASCII hexadecimal digits and can represent a positive, zero, or negative integer. Hexadecimal digits with values 10 through 15 are represented by the ASCII letters a through f or A through F, respectively; each letter used as a hexadecimal digit may be uppercase or lowercase. As in decimal numbers, digits can be split into groups of digits by inserting underscores.

```
HEX_NUMBER      ::= "0" "x" HEX_DIGIT { ["_"] HEX_DIGIT } ;

HEX_DIGIT       ::= "0".."9" | "a".."f" | "A".."F" ;
```

An octal number consists of the leading ASCII characters 0o followed by one or more of the ASCII digits 0 through 7 and can represent a positive, zero, or negative integer. As in decimal and hexadecimal numbers, digits can be split into groups of digits by inserting underscores.

```
OCTAL_NUMBER    ::= "0" "o" OCTAL_DIGIT { ["_"] OCTAL_DIGIT } ;

OCTAL_DIGIT     ::= "0".."7" ;
```

A binary number consists of the leading ASCII characters 0b followed by one or more of the ASCII digits 0 and 1 and can represent a positive, zero, or negative integer. As in decimal, hexadecimal, and octal numbers, digits can be split into groups of digits by inserting underscores.

```
BINARY_NUMBER    ::= "0" "b" BINARY_DIGIT { ["_"] BINARY_DIGIT } ;

BINARY_DIGIT     ::= "0".."1" ;
```

Note that the only unsigned integer literals that can start with a 0 are 0 itself and the hexadecimal, octal, and binary numbers.

Note also that unsigned integer literals are particular instances of the `SPECIAL_IDENTIFIER1` token (see Section 3.6). However, occurrences of underscores are always significant in `SPECIAL_IDENTIFIER1` tokens. For instance, “`1_234`”, “`12_34`”, and “`1234`” denote the same integer literal constant, but distinct identifiers.

Lexically correct integers may be refused by *compilers* if they denote values which do not fit the (implementation dependent) range of type `int` (or `nat`).

A signed integer literal consists of a sign `+` or `-` juxtaposed with a decimal, hexadecimal, octal or binary unsigned integer literal.

3.8.2 Floating-Point Literals

See Section 5.3.4 for a general discussion of the floating-point types and values.

An unsigned floating-point literal may have the following parts: a mandatory whole-number part in decimal notation, an optional decimal point (represented by an ASCII period character) followed by a fractional part, and an optional exponent. The exponent, if present, is indicated by the ASCII letter e or E followed by an optionally signed integer.

```
DIGITS ::= DIGIT { ["_"] DIGIT };

EXPONENT ::= ( "e" | "E" ) [ "+" | "-" ] DIGITS;

REAL ::= DECIMAL_NUMBER "." DIGITS [ EXPONENT ]
      |  DECIMAL_NUMBER EXPONENT;
```

The decimal point must always be preceded and followed by digits. For instance, the lexically incorrect floating-point numbers `4.`, `.12`, and `1.E7` may be written `4.0`, `0.12`, and `1.0E7`, respectively.

The first character of an unsigned floating-point literal can be 0 only if the second character is a decimal point or the letter e or E.

Note that unsigned floating-point literals that do not contain the dot character are particular instances of the `SPECIAL_IDENTIFIER1` token (see Section 3.6). However, occurrences of underscores are always significant in `SPECIAL_IDENTIFIER1` tokens. For instance, “`1_23E4`”, “`12_3E4`”, and “`123E4`” denote the same integer literal constant, but distinct identifiers.

Lexically correct floating point numbers may be refused by *compilers* if they denote values which do not fit (implementation dependent) range of type `float`.

Examples of unsigned floating-point literals:

```
1e1      2.0      0.3      0.0      3.14      6.022137e+23      1e-9
```

A signed floating point literal consists of a sign `+` or `-` juxtaposed with an unsigned floating point literal.

3.8.3 Characters

A character literal is expressed as a character or an escape sequence, enclosed in ASCII single quotes. A character literal is always of type `char`. See Section 5.3.5 for more details on the `char` type.

```
        CHAR ::= "’" CHAR_PRINTABLE "’" ;
        CHAR_PRINTABLE = PRINTABLE | "\"" ;

        PRINTABLE ::= TRULY_PRINTABLE
                    | "\n"     -- linefeed LF
                    | "\t"     -- horizontal tab HT
                    | "\v"     -- vertical tab VT
                    | "\b"     -- backspace BS
                    | "\r"     -- carriage return CR
                    | "\f"     -- form feed FF
                    | "\a"     -- alert BEL
                    | "\\"     -- backslash
                    | "\?"     -- question mark
                    | "\'"     -- single quote '
                    | "\""     -- double quote "
                    | "\\" OCTAL_DIGIT [ OCTAL_DIGIT [ OCTAL_DIGIT ]]
                    | "\\" "x" HEX_DIGIT [ HEX_DIGIT ] ;

        TRULY_PRINTABLE = CHARACTER - "\'\"\\" ;  -- printable characters

        CHARACTER       = #040..#176 + #240..#377 ;
```

The escape sequences allow for the representation of some non graphic characters as well as the single quote, double quote, query, and backslash characters in character literals and string literals.

It is a compile-time error for the character following the `TRULY_PRINTABLE` or `ESCAPE_SEQUENCE` to be other than a ’.

It is a compile-time error for a line terminator to appear after the opening ’ and before the closing ’.

It is a compile-time error if the character following a backslash in an escape is not from the set specified above.

The following are examples of char literals:

```
'a'    '%'    '\t'    '\\'    '\''    '\xFFFF'    '\177'
```

3.8.4 String Literals

A string literal consists of zero or more characters enclosed in double quotes. Each character may be represented by an escape sequence.

A string literal is always of type `string` (§ 5.3.6, p. 34).

```
        STRING ::= "\"" { STRING_PRINTABLE } "\"" ;

        STRING_PRINTABLE = PRINTABLE | "\'" ;
```

As specified in Section (§ 3.2, p. 20), neither of the characters CR and LF is ever considered to be `PRINTABLE`; each is recognized as constituting a line terminator. Instead, one should use the escape sequences “`\n`” for LF and “`\r`” for CR.

It is a compile-time error for a line terminator to appear after the opening ” and before the closing matching ”.

The following are examples of string literals:

```
""    "\""    "\n"   "This is a string"
```

3.9 Operators

The following special symbols are reserved tokens of the languages, formed from ASCII characters. They appear into the concrete syntax between double quotes and in the abstract syntax in teletype fonts.

```
Operator  ::=  "->" | "}" | "]" | ")" | "," | ":" |
            |   ";" | ":=" | "..." | "=" | "==" | "!" |
            |   ">=" | ">" | "<=" | "<" | "|" | "-" | "/"
            |   "!=" | "{" | "[" | "(" | "+" | "**"
            |   "?" | "!?" | "*" ;
```

Chapter 4

Modules

LOTOS NT definitions may be separated into modules, in order to improve code structuration and reuse.

4.1 Module Definition

A module embeds a set of channel, type, function, and process definitions, and may import definitions from other modules.

The syntax of a module definition is the following:

> **module** *mod-id*$_0$ [`(`*mod-id*$_1$`,` ... `,`*mod-id*$_n$`)`]
> [**with** F_0`,` ... `,`F_n] **is**
> *module_pragma*$_1$ ... *module_pragma*$_n$
> MB
> **end module**

where *mod-id*$_0$, ..., *mod-id*$_n$ are module identifiers, $F_0, \dots, F_n$ are predefined function identifiers, *module_pragma*$_1, \dots,$ *module_pragma*$_n$ are module pragmas (see Section 4.2), and MB is a sequence of channel, type, function, and process definitions (see Chapters 5 to 7 and Annex A).

The identifier *mod-id*$_0$ is the name of the current module. The identifiers *mod-id*$_1$, ..., *mod-id*$_n$ are the names of the imported modules.

A module named *mod-id* must be defined in a file named "*mod-id*.lnt", using the same letter case. Otherwise, TRAIAN will issue an error. However, if the module is named TEST (or Test, or test, etc.), the error is replaced by a warning.

The function identifiers $F_0, \dots, F_n$ occurring in the optional **with** clause of the module must be pairwise distinct and be among "`==`", "`!=`", "`<`", "`>`", "`<=`", "`>=`", "`string`", "`ord`", "`card`", "`hash`", "`succ`", and "`pred`".

4.2 Module Pragmas

The syntax of module pragmas is the following:

```
module_pragma  ::=  !int_bits INTEGER                                   (module_pragma1)
                 |  !int_check (0 | 1)
                 |  !int_inf [+ | -] INTEGER
                 |  !int_sup [+ | -] INTEGER
                 |  !nat_bits INTEGER
                 |  !nat_check (0 | 1)
                 |  !nat_inf INTEGER
                 |  !nat_sup INTEGER
                 |  !num_bits INTEGER
                 |  !num_card INTEGER
                 |  !string_card INTEGER
                 |  !update STRING
                 |  !version STRING
```

Chapter 5

Types

LOTOS NT is a strongly typed language, a necessary condition for ensuring description safety.

Type declaration are used to define new types when the few predefined types are insufficient, which is the case of most descriptions. The declaration of new types is very general. However, several well-know type schemes[1] may be derived. In this case, some implicit declarations of other LOTOS NT objects appear.

5.1 Type Definition

A type denotes a domain of values (see Section 2.4) on which LOTOS NT objects are defined.

In LOTOS NT, a type definition must be associated with a name which will be used to refer to it where useful: this association is a type declaration. This means that anonymous types do not exist. For this reason, the equality between types is given by the equality of their names (instead of structural equality).

The definition of types in LOTOS NT follows the general approach of constructed types in functional languages where types are defined using type *constructors*. Constructors are special operations structuring the domain of the type. They give a name to the sub-domain of the type represented by the Cartesian product of the parameters.

The simpler syntax for type definition is the following:

$$\begin{array}{l}
\textbf{type}\ T\ \textbf{is} \\
\quad C_1\ [\texttt{(}V_1^1 \texttt{:} T_1^1, ..., V_{m_1}^1 \texttt{:} T_{m_1}^1\texttt{)}]\texttt{,} \\
\quad \cdots\texttt{,} \\
\quad C_n\ [\texttt{(}V_1^n \texttt{:} T_1^n, ..., V_{m_n}^n \texttt{:} T_{m_n}^n\texttt{)}] \\
\textbf{end type}
\end{array}$$

where $T, T_1^1, ...$ are type identifiers, $C_1, ..., C_n$ are constructor identifiers, and $V_1^1, ...$ are variable identifiers. The default list of parameters is the empty list.

For a constructor C_i, the identifier V_j^i is called *field* or formal parameter, and m_i is called operation arity.

The syntax given above must satisfy the following static semantics constraints:

[1] For LOTOS, these schemes are also known as "rich term syntax" [Pec94].

- There must be at least one constructor declaration ($n \geq 1$).
- For a given constructor C_i, the names of formal parameters must be pairwise distinct, *i.e.*, $\forall j, k \in \{1, ..., m_i\} \quad (j \neq k) \Longrightarrow (V_j^i \neq V_k^i)$.
- For the set of constructors of a given type, fields having the same name should have the same type. For example, the type `HeaderType` defines the values that a header may have:

  ```
  type HeaderType is
      Header1 (dest_id, data_length, header_CRC: nat)
  |   Header2 (dest_id: nat, source_id: nat, data_length, header_CRC: nat)
  entype
  ```

  The field `dest_id` appears in the parameter list of the two constructors with the same type `nat`. Note that the fields having the same type may be grouped in lists, like for `Header1` constructor.
- Two or more constructors may have the same name (may be overloaded) if their profiles (the list of the *types* of parameters and the result type) differ. Note that the name of formal parameters does not solve the overloading.
- 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 iff: (a) it has a constructor of arity 0 or (b) all the parameters of its constructors have productive types.

Example 5.1.1
The type "`bool`" is defined in the (predefined) standard library as an enumeration of two values **true** and **false**, which are the type constructors of arity 0.

```
type bool is
    false,
    true
end type
```

A more elaborate type is the type of a packet which contains a header part and a data part:

```
type PacketType is
    Packet (header: HeaderType, data: DataType)
end type
```

The constructor of type `PacketType`, `Packet`, has two parameters: the first is named `header` and has the type `HeaderType`, the second is named `data` and has the type `DataType`.

A list of packets may be defined using a recursive definition:

```
type PacketListType is
    PacketList_empty,
    PacketList_cons (head: PacketType, tail: PacketListType)
end type
```

The lists may be defined also using the rich term syntax as described in Section 5.4. ■

5.2 Predefined Operations

For each definition of a constructed type T, a set of predefined operations are automatically generated (*sheel* definitions of [BM79]):

- "`==`" and "`!=`", with the profile $T, T \rightarrow$ `bool`, for the equality (*resp.* non equality) test.

- "`<`", "`>`", "`<=`", "`>=`", with the profile $T, T \rightarrow$ `bool`, for the ordering test of values. Note that values of constructed types are ordered lexicographically. The declaration order of constructors is important: the constructor declared first is less than the constructors following it in the declarations.

- "`string`", with the profile $T \rightarrow$ `string`, returns the string representation of the value given as parameter.

- "`ord`", with the profile $T \rightarrow$ `nat`, returns the order number of the (first) constructor of the value.

- "`card`", with the profile $T \rightarrow$ `nat`, returns the number of constructors for T.

Only for finite types (§ 5.4.2, p. 35), the following operations are also defined :

- "`succ`" and "`pred`", with the profile $T \rightarrow T$, return the successor (*resp.* the predecessor) of the value given as parameter. For the border values, these operation are identities.

- "`hash`", with the profile $T \rightarrow$ `nat`, returns the order number of the term in the domain of the type T.

The user may specify explicitly the operations to be automatically generated when the type is declared, using a "**with**" clause:

```
type T is
   ...
   [with op1, ..., opn]
end type
```

where $op_1, ..., op_n$ belong to the set of the predefined operations above. Note that "`==`" and "`!=`" are always generated.

5.3 Predefined Types

As stated in the introduction, a "pure" FDT should not make assumption about the implementation issues. The FDT Lotos respects this constraint by allowing for types like natural numbers or integers only an axiomatic definition. In order to make easier the user task, the standard provides a standard library of data types which contains types like: boolean, natural number, bit, octet, etc.

However, feedback from users showed that the axiomatic definition is not natural and easy to use (e.g., natural numbers where 13 is expressed by 13 compositions of the operation "Succ" applied to "0"!). By consequence, it seems useful to accept natural (programming languages) notations for a set of predefined types. Chapter 3 defines the lexical tokens corresponding to these constants. This alternative definition does not exclude implementation dependent definitions given by the compilers[2].

This section presents some of the predefined types which form the static basis of any Lotos NT description.

[2] For example, Traian provides such an implementation in the file `incl/lotosnt_predefined.h`.

5.3.1 The boolean type

Values of the boolean type, written "bool", are usual truth values **true** and **false**.

Besides the predefined operations provided for usual types, additional operations are available on type bool, e.g., the binary conjunction and disjunction, the unary negation, and comparisons (**false** < **true**). Binary operations may exist in strict and non-strict (short-circuit evaluation) versions. An exhaustive list of these operations is given in Table 5.1.

Name	Profile	Description
not	bool → bool	boolean negation
or	bool, bool → bool	logical disjunction
or else	bool, bool → bool	cancellative or
and	bool, bool → bool	logical conjunction
and then	bool, bool → bool	cancellative and
=>	bool, bool → bool	logical implication
<=>	bool, bool → bool	logical equivalence
xor	bool, bool → bool	exclusive or

Table 5.1: Predefined operations on type bool

5.3.2 The natural type

Values of natural type, written "nat", are natural numbers.

Besides the predefined operations provided for usual types, additional operations available on type nat are, for instance, binary operations such as addition, subtraction, multiplication, (Euclidean) quotient and remainder, and conversions to other numerical types. An exhaustive list of these operations is given on Table 5.2. Operations -, div, mod, int, char, and real can be called without exception parameter, which is equivalent to passing the UNEXPECTED exception.

Name	Profile	May raise	Description
+	nat, nat → nat		addition
-	nat, nat → nat	RANGE_ERROR	subtraction
*	nat, nat → nat		multiplication
**	nat, nat → nat		power
div	nat, nat → nat	ZERO_DIVISION	division
mod	nat, nat → nat	ZERO_DIVISION	modulus
int	nat → int	RANGE_ERROR	integer conversion
char	nat → char	RANGE_ERROR	char conversion
real	nat → real	RANGE_ERROR	real conversion

Table 5.2: Predefined operations on type "nat"

5.3.3 The integral type

Values of integral type, written "int", are signed naturals.

Besides the predefined operations provided for usual types, additional operations available on type int are, for instance, binary operations such as addition, subtraction, multiplication, (Euclidean) division, sign inversion, and conversions to other numerical types. An exhaustive list of these operations is given on Table 5.3. Operations `div`, `mod`, `rem`, `nat`, `char`, and `real` can be called without exception parameter, which is equivalent to passing the `UNEXPECTED` exception.

Name	Profile	May raise	Description
`sign`	`int` → `int`		sign
`-`	`int` → `int`		sign inversion
`+`	`int`, `int` → `int`		addition
`-`	`int`, `int` → `int`		subtraction
`*`	`int`, `int` → `int`		multiplication
`**`	`int`, `nat` → `int`		power
`div`	`int`, `int` → `int`	`ZERO_DIVISION`	division
`mod`	`int`, `int` → `int`	`ZERO_DIVISION`	modulus
`rem`	`int`, `int` → `int`	`ZERO_DIVISION`	remainder
`abs`	`int` → `int`		absolute value
`nat`	`int` → `nat`	`RANGE_ERROR`	natural conversion
`char`	`int` → `char`	`RANGE_ERROR`	char conversion
`real`	`int` → `real`	`RANGE_ERROR`	float conversion

Table 5.3: Predefined operations on type `int`

5.3.4 The floating point type

Values of the floating point type, written "`real`", are signed floating point numbers. Tools may consider implementation defined approximations of real numbers in an implementation-defined range.

Besides the predefined operations provided for usual types, additional operations on these values are the usual arithmetic operations, and conversions to another type. An exhaustive list of these operations is given in Table 5.4. Operations `/` and `int` can be called without exception parameter, which is equivalent to passing the `UNEXPECTED` exception.

Name	Profile	May raise	Description
`-`	`real` → `real`		sign inversion
`abs`	`real` → `real`		absolute value
`+`	`real`, `real` → `real`		addition
`-`	`real`, `real` → `real`		subtraction
`*`	`real`, `real` → `real`		multiplication
`/`	`real`, `real` → `real`	`ZERO_DIVISION`	division
`**`	`real`, `real` → `real`		power
`int`	`real` → `int`	`RANGE_ERROR`	int conversion

Table 5.4: Predefined operations on type `real`

5.3.5 The character type

Values of character type, written "`char`", are characters of the ASCII alphabet. Additional operations available on these values are, e.g., conversion into other types. An exhaustive list of these operations is given in Table 5.5.

Name	Profile	May raise	Description
`nat`	`char` → `nat`		natural conversion
`tolower`, `toupper`	`char` → `char`		conversion
`isupper`, `islower`, `isalpha`, `isdigit`, `isxdigit`, `isalnum`	`char` → `bool`		tests

Table 5.5: Predefined operations on type `char`

NOTE: Different character sets may be considered. Here, we consider for the predefined character type the ISO Latin-1 character set.

5.3.6 The string type

Values of string type, noted "`string`", are dynamic-length character strings.

Additional operations available on these values may be concatenation, getting length of a string, taking a substring of a longer string, taking all of a string except for a substring, inserting a string into another one, searching a given substring in a longer string ... Strings are ordered by lexicographic order. Strings may also be converted to other types. An exhaustive list of these operations is given in Table 5.6. Operations `nat`, `int`, and `real` can be called without exception parameter, which is equivalent to passing the `UNEXPECTED` exception.

Name	Profile	May raise	Description
`length`	`string` → `nat`		length
`&`	`string`, `string` → `string`		concatenation
`index`, `rindex`	`string`, `string` → `nat`		sub-string search
`prefix`, `suffix`	`string`, `nat` → `string`		sub-string selection
`substr`	`string`, `nat`, `nat` → `string`		sub-string selection
`nth`	`string`, `nat` → `char`		the n-th character
`isempty`	`string` → `bool`		emptiness test
`nat`	`string` → `nat`	`RANGE_ERROR`	nat conversion
`int`	`string` → `int`	`RANGE_ERROR`	int conversion
`real`	`string` → `real`	`RANGE_ERROR`	real conversion

Table 5.6: Predefined operations on type `string`

5.4 Derived Types

This section presents how some derived type declarations are introduced as syntactic sugar of the more general type declaration.

5.4.1 Enumerated types

The *enumerated* type definition declares values of a (finite) domain ordered by the declaration order. The declaration of an enumerated type T_E with values C_1, ..., C_n has the following syntax:

```
type $T_E$ is
  $C_1 , ... , C_n$
  [with $op_1, ..., op_n$]
end type
```

For example:

```
type day_of_week is
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
    with <
end type
```

The order in which the values of the type are declared induces an order relation, for example `Monday` < `Wednesday`.

The declaration above is translated into the following constructed type declaration:

```
type $T_E$ is
  $C_1$  ,  ...  ,  $C_2$
  [with $op_1, ..., op_n$]
end type
```

5.4.2 Scalar and simple types

Scalar types are `bool`, `nat`, `int`, `char`, user defined *finite* types and *enumerable* types, and those only.

A type is *finite* if its domain is finite. Informally, a type is finite if either its constructors are of arity 0 (enumerated types), or the arguments of its constructors are finite types which do not depend recursively on the current type.

Formally, it is possible to detect finite types by constructing the dependency graph between types. A type T depends on type T' if T' appears as the type of an argument of a constructor of type T. The finite types are those contained in acyclic sub-graphs (trees) of the dependency graph having as leaves user defined enumerated types, or `bool`, or `char`.

An *enumerable* type is a type whose domain is isomorphic with the domain of natural numbers, and a total order relation defined on its elements. For the user defined types, the order relation is given by the lexicographic order induced by the declaration order of the constructors. Consider for example the type `HeaderType` (§ 5.1, p. 29). Values constructed using the `Header1` constructor are smaller than values constructed using the constructor `Header2`.

For enumerable types, it is possible to define the functions `init` and `succ`, but not `max`.

Simple types are scalar types plus the type `real`.

5.4.3 Record types

A *record* type corresponds to the Cartesian product of component types, except that component values are accessed by a name rather than by their position.

A record type definition T_R with fields V_1 of type T_1, ..., V_n of type T_n can be defined as follows:

> **type** T_R **is**
> T_R`(`V_1`:`$T_1, ..., V_n$`:`T_n`)`
> [**with** $op_1, ..., op_n$]
> **end type**

Operations such as selection of a field, equality, inequality, and comparisons are defined as for the constructed types.

5.4.4 Lists

Values of type *list* are ordered, linear lists, the elements of which belong to the same type, called the *element* type. There is no restriction on this type.

The definition of a list type T_L with elements of type T has the following syntax:

> **type** T_L **is**
> **list of** T
> [**with** $op_1, ..., op_n$]
> **end type**

This definition is translated into a constructed type definition as follows:

> **type** T_L **is**
> `NIL, CONS(HEAD:`T`,TAIL:`T_L`)`
> [**with** $op_1, ..., op_n$]
> **end type**

Note that the recursive type definition "**type** T **is list of** T **end type**" is correct. It is expanded into "**type** T **is** `NIL, CONS (HEAD :` T`, TAIL :` T`)` **end type**", which is the definition of a binary tree.

An exhaustive list of these operations for a type T_L (list of elements of type T) is presented in Table 5.7.

Name	Profile	May raise	Description
`nil`	$\rightarrow T_L$		empty list
`cons`	$T, T_L \rightarrow T_L$		list cons (front)

Table 5.7: Operations predefined for a list type

The list may be specified "in extenso" by using the following notation:

> `{`$E_1, ..., E_n$`}`

where E_1, ..., E_n are expressions returning values of type T. This notation is equivalent to `CONS(`E_1`,CONS(`E_2`,...,CONS(`E_n`,NIL)))`.

5.5 Sorted lists

Sorted lists are accepted by the parser but not yet fully implemented.

The definition of a sorted list type T_L with elements of type T has the following syntax:

> **type** T_L **is**
>   **sorted list of** T
>   [**with** $op_1, ..., op_n$]
> **end type**

5.6 Sets

Sets are accepted by the parser but not yet fully implemented.

The definition of a set type T_S with elements of type T has the following syntax:

> **type** T_S **is**
>   **set of** T
>   [**with** $op_1, ..., op_n$]
> **end type**

5.7 Sorted sets

Sorted sets are accepted by the parser but not yet fully implemented.

The definition of a sorted set type T_S with elements of type T has the following syntax:

> **type** T_S **is**
>   **sorted set of** T
>   [**with** $op_1, ..., op_n$]
> **end type**

5.8 Arrays

Arrays are accepted by the parser but not yet fully implemented.

The definition of an array type T_A with elements of type T has the following syntax:

> **type** T_A **is**
>   **array** `[`*array_bound*`..`*array_bound*`]` **of** T
>   [**with** $op_1, ..., op_n$]
> **end type**

Array bounds are non-negative integers defined by the following grammar:

array_bound	::=	`INTEGER`	*unsigned integer*	(array_bound1)

5.9 Ranges

Ranges are accepted by the parser but not yet fully implemented.

The definition of a range type T_R with elements of type T has the following syntax:

type T_R **is**
  range *range_bound* . . *range_bound* **of** T
  [**with** $op_1, ..., op_n$]
end type

Range bounds are either characters or (possibly signed) integers defined by the following grammar:

range_bound	::=	`INTEGER`	*unsigned integer*	(range_bound1)
	|	(`+` | `-`) `INTEGER`	*signed integer*	(range_bound2)
	|	`CHAR`	*character*	(range_bound3)

5.10 Predicate types

Predicate types are accepted by the parser but not yet fully implemented.

The definition of a predicate type T_P with elements of type T satisfying a predicate E has the following syntax:

type T_P **is**
  V : T **where** E
  [**with** $op_1, ..., op_n$]
end type

E must be a Boolean value expression (see Section 6.2 page 42) which may use the variable V.

5.11 External Types and Pragmas

In order to interface with other languages, a type definition may specify the name which should be used for the type in an implementation. Moreover, it can also specify that a type has an external definition. This is done using *pragmas*.

The general syntax for type definition becomes:

type T **is** *type-pragmas*
  C_1 [$(\overline{V}^1_1 : T^1_1, ..., \overline{V}^1_{m_1} : T^1_{m_1})$] *operation-pragmas* **,**
  $\cdots$ **,**
  C_n [$(\overline{V}^n_1 : T^n_1, ..., \overline{V}^n_{m_n} : T^n_{m_n})$] *operation-pragmas*
 [**with** $op_1, ..., op_n$]
end type

Types pragmas are lists of *type-pragma* having the following forms:

- "`!external`" if the type has an external implementation; this implementation should be provided in an external file having the extension "`.tnt`".
- "`!implementedby "`*name*`"`" (or equivalently "`!implementedby "C:`*name*`"`") if the external C name used for the type is *name*.
- "`!comparedby "`*name*`"`" (or equivalently "`!comparedby "C:`*name*`"`") if the equality function `==` should be implemented by a C function named *name*.

- “`!printedby "`*name*`"`” (or equivalently “`!printedby "C:`*name*`"`”) if the values of the type should be printed by a C function named *name*.
- “`!iteratedby "`*name_1*`", "`*name_2*`"`” (or equivalently “`!iteratedby "C:`*name_1*`", "C:`*name_2*`"`”) if the values of the type should be iterated (if possible) by a couple of C functions: *name_1*, which has no parameter and provides the first value of the type, and *name_2*, which takes as parameter a defined variable and either assigns it the next value and returns true, or returns false if the input parameter value was the last value of the type.
- “`!pointer`” if the type has to be implemented by a pointer in C.
- “`!nopointer`” if the type does not have to be implemented by a pointer in C. If this constraint cannot be satisfied (e.g., because the type is self-recursive, or because it is part of a set of mutually-recursive types, all of which have the pragma “`!nopointer`”), then Traian issues an error.
- “`!card` n”, where $n > 1$, if all values of type T have to be stored into a hash table of size n and, thus, represented as entries within this table. The hash table is extensible, meaning that T can have more than n different values. For performance reasons, it is advised to choose a value of n close to the cardinal of T.
- “`!bits` n”, where $n > 0$, has the same meaning as “`!card` 2^n”.
- “`!list`” if the type is isomorphic to a list type and should be printed in the braced form “$\{v_1, \ldots, v_n\}$” rather than as constructors with parameters.

The type pragmas “`!pointer`”, “`!nopointer`”, “`!card` n”, and “`!bits` n” are mutually exclusive. They cannot be given to singleton types and enumeration types. The type pragmas “`!pointer`” and “`!nopointer`” cannot be given to numeral types.

Type pragmas of the form “`!implementedby "LOTOS:`*name*`"`”, “`!comparedby "LOTOS:`*name*`"`”, “`!printedby "LOTOS:`*name*`"`”, and “`!iteratedby "LOTOS:`*name_1*`", "LOTOS:`*name_2*`"`” are accepted, but for the moment they are ignored silently.

Operation pragmas are lists of *operation-pragma* having the following forms:

- “`!external`” if the operation has an external implementation in a file having the extension “`.fnt`”.
- “`!implementedby "`*name*`"`” (or equivalently “`!implementedby "C:`*name*`"`”) if the external name of the operation is *name*.

Operation pragmas of the form “`!implementedby "LOTOS:`*name*`"`” are accepted, but for the moment they are ignored silently.

In pragmas of the form “`!comparedby "`*name*`"`”, “`!comparedby "C:`*name*`"`”, “`!implementedby "`*name*`"`”, “`!implementedby "C:`*name*`"`”, “`!iteratedby "`*name_1*`", "`*name_2*`"`”, “`!iteratedby "C:`*name_1*`", "C:`*name_2*`"`”, “`!printedby "`*name*`"`”, and “`!printedby "C:`*name*`"`”, *name*, *name_1*, and *name_2* 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”).

Chapter 6

Expressions, Statements, and Functions

The data part of LOTOS NT is mainly based on types, expressions, statement, and functions. Types and type definitions are presented in Chapter 5; expressions, statements, and functions are presented in this chapter.

One may replace the data part of LOTOS NT with another data description formalism ensuring the safety property, for example ACTONE data types. Note that the behaviour part of the language contains symmetrical constructs of the data part.

There is a fundamental difference between the expressions/statements (and functions) and behaviours (and processes). An expression or a statement cannot contain communication, non-determinism, concurrency, or real-time. The data part is a sequential and deterministic language. The evaluation of an expression takes no time, and should return a value, or may raise an event. The evaluation of a statement takes no time, and may return a value, assign some variables, and raise an event.

An important characteristic of the data language presented here is its "clean" imperative style. The language supports assignment and other imperative style facilities, but a proper semantics is given [Sig99], which restricts undesirable effects like the use of uninitialized variables. The imperative style is combined with constructs specific of functional languages, e.g., *pattern-matching*.

This chapter presents all aspects related to data language: constants, variables, expressions, statements, and function declarations.

6.1 Constants

Primitive constants, written K, are boolean values **true** and **false**, signed or unsigned integer literals (of type `int` or `nat`), signed or unsigned floating point literals (of type `real`), character literals (of type `char`), and string literals (of type `string`). See Section 3.8 for details about the syntax of literals.

Literals cannot be followed by parentheses. For instance, "`1 ()`" is allowed only if a nullary function or constructor named `1` is defined. In that case it represents a call to this function or constructor, rather than the natural or integer constant 1.

6.2 Value expressions

Value expressions (or shortly expressions) are primitive constants and terms of the user defined types built using the application of constructors or functions. As a consequence, any value is typed. Moreover, value expressions cannot assign variables and may raise events (§ 6.4.9, p. 56) only by function calls.

The following grammar gives the syntax of value expressions. Value expressions followed by a star (*) are not yet fully implemented.

$$\begin{array}{lcllr}
E & ::= & K & \textit{primitive constant} & \text{(E1)}\\
 & | & V & \textit{variable} & \text{(E2)}\\
 & | & V\texttt{.}\textbf{in} & \textit{input argument (in postcondition)} & \text{(E3)}\\
 & | & V\texttt{.}\textbf{out} & \textit{output argument (in postcondition)} & \text{(E4)}\\
 & | & \textbf{result} & \textit{function result (in postcondition)} & \text{(E5)}\\
 & | & C\ [\texttt{(}ES\texttt{)}] & \textit{constructor application} & \text{(E6)}\\
 & | & E\ C\ E & \textit{infix constructor application} & \text{(E7)}\\
 & | & '\{'\ [E\ \{\texttt{,}E\}]\ '\}' & \textit{list or set construction} & \text{(E8)}\\
 & | & F\ [\texttt{(}ES\texttt{)}] & \textit{function call} & \text{(E9)}\\
 & | & F\ \texttt{[}XS\texttt{]}\ \texttt{(}ES\texttt{)} & \textit{function call with exceptions} & \text{(E10)}\\
 & | & E\ F[\texttt{[}XS\texttt{]}]\ E & \textit{infix function call} & \text{(E11)}\\
 & | & E\ \texttt{[}E\texttt{]} & \textit{array access*} & \text{(E12)}\\
 & | & E\texttt{.}[\texttt{[}X\texttt{]}]V & \textit{field selection} & \text{(E13)}\\
 & | & E\texttt{.}[\texttt{[}X\texttt{]}]'\{'\ ES\ '\}' & \textit{field update} & \text{(E14)}\\
 & | & E\ \textbf{of}\ T & \textit{type coercion} & \text{(E15)}\\
 & | & \texttt{(}E\texttt{)} & \textit{parenthesized expression} & \text{(E16)}
\end{array}$$

$$\begin{array}{lcllr}
ES & ::= & & \textit{empty record} & \text{(ES1)}\\
 & | & V_1\texttt{->}E_1\texttt{,}\ldots\texttt{,}V_n\texttt{->}E_n & \textit{record} & \text{(ES2)}\\
 & | & E_1\texttt{,}..\texttt{,}E_n & \textit{tuple} & \text{(ES3)}
\end{array}$$

$$\begin{array}{lcllr}
XS & ::= & [\texttt{...}] & \textit{empty or dotted record} & \text{(XS1)}\\
 & | & X'_1\texttt{->}X_1\texttt{,}\ldots\texttt{,}X'_n\texttt{->}X_n\ [\texttt{,...}] & \textit{record} & \text{(XS2)}\\
 & | & X_1\texttt{,}..\texttt{,}X_n & \textit{tuple} & \text{(XS3)}
\end{array}$$

where $E, E_1, \ldots$ denote value expressions, V denotes variables, C denotes constructor identifiers, ES denotes lists of actual value expression parameters (named for record or unnamed for tuple), and XS is a list of actual event parameters (§ 6.4.9, p. 56).

In addition to the above syntactic rules, calls to functions and constructors whose identifier is a special identifier (see Section 3.6) have to satisfy the following:

- A nullary function or constructor (i.e., a function or constructor that has no variable parameter) whose identifier is a SPECIAL_IDENTIFIER1 must be called using parentheses if it has event parameters. For instance, one must write “7T [E] ()” instead of “7T [E]”. However, if such a function or constructor has no event parameters, then it can be used without parentheses.

- A nullary function or constructor whose identifier is a SPECIAL_IDENTIFIER2 must always be called using parentheses, even if it has no event parameters. For instance, one must write “<> ()” instead of “<>” and “% [E] ()” instead of “% [E]”.
- A unary function or constructor (i.e., a function or constructor that has a single variable parameter) whose name is a SPECIAL_IDENTIFIER2 can be called without enclosing its argument within parentheses. For instance, one can write “-X” instead of “- (X)” and “& [E_1 , E_2] 7” instead of “& [E_1 , E_2] (7)”.
- A binary function or constructor (i.e., a function or constructor that has two variable parameters) whose identifier is a SPECIAL_IDENTIFIER1 cannot be used in the infix form.

The precedence of operators appearing in expressions is given in table 6.2. The letter case of identifiers appearing in this table must be respected for the given precedence to be taken into account. Otherwise, a warning is issued.

Priority	*Operations*
0.	. field selection and update, [...] array access
1.	unary operators
2.	**of**
3.	infix binary operators not listed below
4.	**
5.	*, /, div, mod, rem
6.	+, -
7.	==, =, !=, <, <=, >=, >
8.	and, and then, or, or else, xor, =>, <=>

Note that **of** expressions are not allowed under . field selection and update, unless the **of** expression is enclosed within parentheses. For instance, “X **of** T.Y” is not allowed and should be written “(X **of** T).Y” instead.

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., functions of priority level 2 in Table A.2).

All (infix) operators of same precedence are parsed from left to right, meaning that “E_1 op_1 E_2 op_2 E_3” is parsed as “(E_1 op_1 E_2) op_2 E_3” rather than “E_1 op_1 (E_2 op_2 E_3)”.

The precedence of operators given in table A.2 changed in February 2021 (Traian version 3.3 released simultaneously with CADP version 2021-b). Since then, warnings are triggered whenever expressions are parsed differently due to this change. The module pragma “!update "2021-b"” can be used to declare that the new precedence of operators is taken into account, and then avoid these warnings. The scope of this pragma is not the entire module, but only the file in which it occurs.

Value expressions are evaluated to *value* or normal forms. Values are primitive constants and ground terms of the user defined types built using the application of constructors on value records. We will write values $N, N_1, ...$ and sequences of values NS.

The construct "`...`" for event parameters allows the user to avoid the explicit instantiation of the actual event parameters if they are the same name as the formal parameters. For example, if the formal event parameter of the function F is called X, one may call F like "F `[...] ()`" which is replaced by the compiler with "F `[`X`->`X`] ()`"; in this case, X should be already declared in the environment.

The expression "V`.`**in**" can be used only in postconditions of functions that contain an "**in**", "**in var**", or "**in out**" parameter V. Its value is the value of parameter V when entering the function in the postcondition of which it occurs.

The expression "V`.`**out**" can be used only in postconditions of functions that contain an "**out**", "**out var**", or "**in out**" parameter V. Its value is the value of parameter V when exiting the function in the postcondition of which it occurs.

The keyword **result** can be used only in postconditions of functions that return a result. Its value is the result returned by the function in the postcondition of which it occurs.

6.2.1 Variables

Variables, noted V, are *assignable* objects containing values which are computed elsewhere. Note that, from this point of view, the LOTOS NT data language is an imperative language: it supposes the existence of a *memory* (a set of cells represented by variables which can store some values) which can be accessed for read and write operations. However, the static semantics constraints impose a clean imperative style: the access to an uninitialized cell (variable) is signaled at compile time and does not produce a run-time error.

A value expression may be a variable V. The variable must be initialized (contains a value). The type of the expression is the type of the variable, and the result of the expression evaluation is the value of the variable V.

6.2.2 Constructor application

The constructor application computes values of the domain of their target type. The constructor should be already defined in the current environment by type definitions (§ 5.1, p. 29). The actual list of arguments of the constructor may be expressed either *by name* giving a record whose fields are labelled with the names of formal parameters (alternative ES2), or *by position* giving the (ordered) list of actual values (alternative ES3).

The expressions below use positional constructor application:

C `(`E_1`, ..., `E_n`)`
E_1 C E_2

the following expression use the named style:

C `(`V_1`->`E_1`, ..., `V_n`->`E_n`)`

Note that the positional style may be translated into the named style using the static semantics informations about the constructor declaration.

The number of actual parameters must be the same as the arity of the constructor. In the named style, the names of the formal parameters (fields $V_1, ..., V_n$) must be pairwise distinct, *i.e.*, a formal parameter V_i should appear only once in the list. The values $E_1, ..., E_n$ associated with these names must have the same types as the corresponding formal parameters.

If the constructor has only two parameters it can be applied in the infix positional style.

If the constructor is overloaded, the informations given by the type of its parameters and the type of the resulting value should suffice to solve the overloading (*i.e.*, to find the unique constructor having this profile).

The evaluation of constructor application begins with the left-to-right evaluation of its actual parameters. The values obtained are used to form the constructed value which is the result of evaluation. If one expression E_i raises an event (§ 6.4.9, p. 56), the evaluation is blocked, and the event is signaled.

Example 6.2.1
`Monday` is a value of the type `day-of-week`.

`Header1 (1, 2, 3)`, `Header2 (1, 0, 2, 3)` are values of type `HeaderType`.

`Header1 (1, 2, 3)` and `Header1 (data_length -> 2, dest_id -> 1, header_CRC -> 3)` represent the same value of type `HeaderType`. ■

6.2.3 Function application

The function application is largely treated in Section 6.5.2. Note that, for function application expressions the function should return a value and cannot do side effects (have only "**in**" and/or "**in var**" parameters).

Functions may be predefined operations described in 5.3.

6.2.4 Field selection

A field selection expression has the general form:

E`.[`X`]`V

where E is an expression and its value is of the form C`(`V_1`->`$N_1, ..., V_n$`->`N_n`)` (*i.e.*, a constructed value) and V is one of the fields $V_1, ..., V_n$. The selection expression returns the value of this field.

It is worth noticing that the event X is raised (see Section 6.4.9) if the value returned by E does not have a field of name V. In fact, the static semantics ensures that the field V is a field of one of the constructors of the E type. However, this does not suffice to ensure that no dynamic error arises.

NOTE: To be sure that no event is raised, one may wish to use field selections for record types only. In this case, the static semantics will ensure that the single constructor of the record type has the field as argument; thus, the raise clause should be omitted with current version of the compiler.

Example 6.2.2
If E is an expression of type `HeaderType`, the following expression:

E`.[`X`]dest_id`

selects the component "`dest_id`" of the value. More precisely, if E evaluates to "`Header1 (data_length -> 2, dest_id -> 1, header_CRC -> 3)`", the selection expression above returns 1; if E evaluates to "`Header2 (1, 0, 2, 3)`", the selection expression evaluates to 1 (the field "`dest_id`" is the first parameter of the "`Header2`" constructor). ■

The field selection may be translated into a function call. For example, if E is an expression of type T below:

```
**type** $T$ **is**
    $C_1(V_1 : T_1)$
  | $C_2(V_2 : T_2)$
  | $C_3(V_1 : T_1, V_2 : T_2)$
  | $C_4(V_4 : T_4)$
**end type**
```

then, the expression E`.[`X`]`V_1 is equivalent to a function call having the following body:

```
**case** $E$ **in**
    $C_1(V_1 : T_1)$
  | $C_3(V_1 : T_1, V_2 : T_2)$ -> **return** $V_1$
  | **any** $T$ -> **raise** $X$
**end case**
```

6.2.5 Field update

A field update expression has the general form:

E`.`[`[`X`]`]$\{V_1$`->`E_1`, ..., `V_n`->`$E_n\}$

where E is an expression, X is an event, and the value of E is of form C`(`V'_1`->`$N_1, ..., V'_m$`->`N_m`)` (*i.e.*, a constructed value) and $\{V_1, ..., V_n\} \subseteq \{V'_1, ..., V'_m\}$.

The update expression returns the value of E where the fields $V_1, ..., V_n$ have been modified to values resulted from the evaluation of $E_1, ..., E_n$ expressions. If the value represented by E has not (all) the fields $V_1, ..., V_n$, the event X is raised.

NOTE: To be sure that no event is raised, one may wish to use field updates only for types all constructors of which have fields $V_1, ..., V_n$. In this case, the static semantics will ensure that no event can be raised, and the event should be omitted with current version of the compiler.

6.2.6 Unambiguous Expressions

To solve the type ambiguity introduced by the function and constructor overloading, the explicit typing of expression is allowed:

E **of** T

The evaluation of this expression is the same as the evaluation of E; the type T is used only at compile time.

Another source of ambiguity is the precedence of LOTOS NT predefined operations. This precedence can be forced using parenthesized expressions:

`(`E`)`

(E) evaluates like E. Parenthesis may be also used for esthetic considerations.

6.3 Patterns

A *pattern* is a construct allowing to obtain informations about the structure of values. The patterns, denoted by P, have the following form:

P	$::=$	V	*variable*	(P1)
	$\mid$	K	*constant pattern*	(P2)
	$\mid$	**any** $[T]$	*wildcard*	(P3)
	$\mid$	V **as** P	*aliasing*	(P4)
	$\mid$	$C\ [\texttt{(}PS\texttt{)}]$	*constructed pattern*	(P5)
	$\mid$	$P\ C\ P$	*constructed pattern infixed*	(P6)
	$\mid$	$'\{'PS'\}'$	*list pattern*	(P7)
	$\mid$	P **of** T	*explicit typing*	(P8)
	$\mid$	P **where** E	*guarded pattern*	(P9)
	$\mid$	$\texttt{(}P\texttt{)}$	*parenthesized pattern*	(P10)

where E is an expression of type `bool` and PS denotes lists of pattern parameters (named for records or unnamed for tuples):

PS	$::=$	$[\texttt{...}]$	*empty or wildcard record*	(PS1)
	$\mid$	$V_1\texttt{->}P_1, ..., V_n\texttt{->}P_n\ [\texttt{,...}]$	*record*	(PS2)
	$\mid$	$P_1, ..., P_n$	*tuple*	(PS3)

The additional rules for calls to unary and nullary constructors whose identifier is is a special are the same as for functions in expressions.

The precedence of infix constructors in patterns is the same as in table 6.2, except "`and then`" and "`or else`", which are not permitted. The "**as**" construct has the lowest precedence, i.e., "V **as** $P_1\ C\ P_2$" is parsed as "V **as** $(P_1\ C\ P_2)$" for any infix constructor C.

Due to the coexistence of both forms "**any**" and "**any** T", some patterns may have an ambiguous interpretation. For instance, "**any** $X\ Y\ (Z)$" could have the following two interpretations:

1. X might be interpreted as a type and therefore, Y as an infix constructor and (Z) as a pattern between parentheses, or

2. X might be interpreted as an infix constructor and therefore, "$Y\ (Z)$" as a pattern consisting of a constructor Y and a pattern parameter Z.

To avoid such ambiguities, TRAIAN requires that the first symbol following T in "**any** T" is not an identifier (neither normal nor special). As a consequence, "**any** T" must be enclosed in parentheses when used on the left of an infix constructor, e.g., one should write "`(`**any** T`)` $C\ P$" instead of "**any** $T\ C\ P$". The correct interpretation of "**any** $X\ Y\ (Z)$" is therefore the second one in the above enumeration. Note that "**any** T **of** T" does not require parentheses around "**any** T", as **of** is a keyword.

The variables V belonging to a pattern P are "initialization" occurrences (*i.e.*, they should be already declared, but may be non initialized). It is not allowed to use several times the same variable V in the same pattern P.

NOTE: Like in ActOne, and unlike in functional languages, the occurrence of a variable in a pattern is a "use" occurrence and not a "define" occurrence. This design choice is compatible with the "declare before use" requirement of imperative style languages.

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

1. Sends a boolean result which is true if N has the same structure as P, false otherwise.
2. If N matches P, the variables V used by P are initialized with the values extracted from N.

Matching is defined (recursively) as follows. Remind that patterns and values match only if they have the same type.

Pattern	Value	Condition	Effect	Result
V	N	None	V receives N	true
K	K	None	None	true
K	N	$K \neq N$	None	false
any	N	None	None	true
V **as** P	N	P and N match	V receives N	true
V **as** P	N	P and N do not match	None	false
$C(P_1, \ldots, P_n)$	$C(N_1, \ldots, N_n)$	Each P_i, N_i match	None	true
$C(P_1, \ldots, P_n)$	$C(N_1, \ldots, N_n)$	Some P_i, N_i do not match	None	false
$C(P_1, \ldots, P_n)$	N	N has not the form $C(N_1, \ldots, N_n)$	None	false
P **of** T	N	Same as matching P and N		
P **where** E	N	Matching P and N returns true and E evaluates to `true`	Same as matching P and N	true
P **where** E	N	Matching P and N returns false or E evaluates to `false`	None	false

Note that:

- in the pattern V **as** P, V cannot occur in P;
- P_1 C P_2 is defined exactly like $C(P_1, P_2)$;
- P **of** T is used only to solve ambiguities caused by constructor overloading;
- For P **where** E, evaluation of E takes into account the effect of matching P and N, i.e., evaluation of E takes places in the context of variables bound by the matching.

The "**...**" notation is a shorthand meaning that all fields of the record have an "**any**" pattern. Note that the type of the record should be unambiguous. Also, the "**...**" shorthand can be used following a sequence of labelled patterns. It will be translated to a record in which the unspecified fields are "**any**" patterns.

Example 6.3.1
For a value of type `HeaderType`, the following patterns:

```
Header1 (dest, length, crc)
Header2 (dest_id -> dest of int, data_length -> data, header_CRC -> crc, ...)
```

allow to obtain the destination (into the variable `dest`), the length of data (into the variable `length`), and the CRC (into the variable `crc`) of the value. The source value is neglected since the "`...`" are translated to `source_id -> any`. ■

Note that the variables initialized by the matching of the pattern P against the value N may be used in the remainder of the description iff the pattern-matching is successful. This ensures that the variables defined inside a pattern are always initialized before use.

The patterns are mainly used in the "**case**" statement (§ 6.4.6, p. 52).

6.4 Statements

A Lotos NT *statement*, denoted by I, may return a value, assign variables, and raise events. The main difference between expressions and statements is that statements only may assign variables and explicitly raise events.

Each statement is typed by the record of variables assigned and the value returned. The evaluation of a statement may modify the memory, return a value, or/and raise an event.

The following grammar gives the syntax of statements. Statements followed by a star (*) are not yet fully implemented.

I	::=	**return** E	*value return*	(I 1)
	|	**null**	*termination*	(I 2)
	|	V`:=`E	*assignment*	(I 3)
	|	V `[`E`]:=`E	*array assignment**	(I 4)
	|	I `;` I	*sequential composition*	(I 5)
	|	**var** $\overline{V}$`:`T {`,`$\overline{V}$`:`T} **in** I **end var**	*variable declaration*	(I 6)

	|	**case** E {`,`E} [**var** VL] **in**
		IM
		end case
			case statement	(I 7)

	|	**if** E **then** I
		{ **elsif** E **then** I }
		[**else** I]
		end if
			conditional statement	(I 8)

	|	[**eval**] [V`:=`] F [`[`XS`]`] [`(`VS`)`]	*procedure call*	(I 9)
	|	**loop** I **end loop**	*forever loop*	(I 10)
	|	**loop** X **in** I **end loop**	*breakable loop*	(I 11)

	|	**while** E **loop**
		I
		end loop
			while loop	(I 12)

	|	**while** E **loop** X **in**
		I
		end loop
			breakable while loop	(I 13)

	|	**for** I **while** E **by** I **loop**
		I
		end loop
			for loop	(I 14)

	|	**for** I **while** E **by** I **loop** X **in**
		I
		end loop
			breakable for loop	(I 15)

	|	**break** X	*loop break*	(I 16)
	|	**raise** X [`()`]	*raise event*	(I 17)
	|	**assert** E [**raise** X [`()`]]	*assertion*	(I 18)
	|	**trap** {**exception** X **is** I} **in** I **end trap**	*trapping events*	(I 19)
	|	**use** $V\{,V\}$	*variable use*	(I 20)
	|	**access** $X\{,X\}$	*event access*	(I 21)

$\overline{V}$	::=	$V\ \{,V\}$	*list of variable identifiers*	(VL1)
VL	::=	$\vec{V}:T\ \{,\vec{V}:T\}$	*variable list*	(VL2)

IM	::=	P {,P} {`|` P {,P}} `->` I	*match-statement*	(IM1)
	|	IM `|` IM	*list*	(IM2)

VE	::=	E	*actual parameter "in" or "in var"*	(VE1)
	|	`?`V	*actual parameter "out" or "out var"*	(VE2)
	|	`!?`V	*actual parameter "in out"*	(VE3)

VS	::=	[`...`]	*empty or wildcard*	(VS1)
	|	$V \Rightarrow VE\ \{,V \Rightarrow VE\}$ [`,...`]	*disjoint union*	(VS2)
	|	$VE\ \{,VE\}$	*list*	(VS3)

where IM are match instructions, VE are actual value parameter, and VS are sequences of actual value parameters.

In the following we present each Lotos NT statement.

6.4.1 Value return

The evaluation of the statement "**return** E" begins with the evaluation of E. If E evaluates successfully, the statement returns the value of E. If E raises an event, the statement raises the same event and terminates unsuccessfully (blocks).

6.4.2 Null Statement

The statement "**null**" has no other effect than termination. It does not return any value and it does not assign any variable.

6.4.3 Assignment

The effect of an assignment statement "V`:=`E" is the modification of the value stored by the variable V at the value given by the expression E. Note that side effects are avoided because the expression E cannot assign other variables. It can only return a value.

The evaluation starts by evaluating E. If E terminates successfully, the resulting value is assigned to the variable V. If E raises an event (§ 6.4.9, p. 56), the event is propagated and V is not assigned.

The value of a variable may also be modified by function call (§ 6.5.2, p. 61). A variable can take several successive values, for example:

```
V := 0 ; V := V + 1
```

where the variable `V` receives values 0 and 1. As long as statements and expressions cannot have a behaviour, the variable `V` takes these values at the same instant.

6.4.4 Sequential Composition

In the sequential composition of statements "I_1 `;` I_2", the statement I_1 cannot return a value but may assign variables (*i.e.*, the return statement should be in the final position of the sequential composition).

The evaluation starts by evaluating I_1. If I_1 terminates successfully, the result is given by the evaluation of I_2. If I_1 raises an event (§ 6.4.9, p. 56), the event is propagated and I_2 is never started. For example "`raise X; V:=1`" will never assign 1 to `V`.

6.4.5 Variable declaration

Variables may be declared using the local variable declaration statement, which has the simple form:

var $V_1 : T_1, ..., V_n : T_n$
in I
end var

where $V_1, ..., V_n$ are variable identifiers, $T_1, ..., T_n$ are type identifiers, and I is a statement.

A "**var**" statement declares the names of variables having the same scope, and their types. The scope of a variable declaration is the body I. Scoping is lexical: any re-declaration of a variable hides the outer declaration.

Variables $V_1, ..., V_n$ should be different.

6.4.6 Case statement

Lotos NT allows to describe conditional processing of data by using constructs similar to those used by the usual programming languages: “**case**” and “**if**”.

The most general conditional statement offered by Lotos NT is the “**case**” statement, whose simplest form is:

```
case $E_0$
   var $V_1 : T_1 , ... , V_n : T_n$ in
   $P_1$ -> $I_1$
 | ...
 | $P_n$ -> $I_n$
end case
```

where $n \geq 1$. The expression E_0 must have the same type as the patterns $P_1, ..., P_n$. The statements $I_1, ..., I_n$ should return a value of the same type and initialize the same set of (non-local declared) variables. This condition is important for the control of the variable flow. The scope of variables $V_1, ..., V_n$ are the patterns P_i and the statements I_i.

The patterns $P_1, ..., P_n$ must be exhaustive, *i.e.*, they must cover all the possible values of type T. There exists algorithms that check statically this condition [Sch88]. To make a list of patterns exhaustive one can add a clause “**any of** T `->` I_{n+1}” at the end of the list.

The evaluation of a “**case**” statement is made as follows. Let N_0 be the value of the expression E_0; N_0 is matched sequentially over the clauses corresponding to patterns $P_1, ..., P_n$ until it matches one. The result of the case statement is the same as the result of the statement I_i (evaluated in the context of variables bound by P_i) corresponding to the first clause i which matches N_0.

Example 6.4.1
The statement below returns the destination identifier of an expression E of type `HeaderType`:

```
case E of HeaderType
  var dest: int in
    Header1 (dest, any, any) -> return dest
|   Header2 (dest -> dest, ...) -> return dest
end case
```

Note that the patterns cover all the values of type `HeaderType`. The wildcard “`any`” are used to match all values which are not interesting for the remainder of the statement. ■

Note that constant values may be filtered by giving their values, excepting the floating point values (of `real` type). Constants may also be filtered by using the “**if**” statement.

A more sophisticated form of the “**case**” statement provides factorization of clauses which have the same target statement I_i. Consider for example, the statement which encodes the working days of a week by 1 and the week-end days by 0 using the above “**case**” statement:

```
case E in
    Monday    -> return 1
|   Tuesday   -> return 1
|   Wednesday -> return 1
|   Thursday  -> return 1
|   Friday    -> return 1
|   Saturday  -> return 0
|   Sunday    -> return 0
end case
```

A simpler form with the same effect is:

```
case E in
    Monday | Tuesday | Wednesday | Thursday | Friday -> return 1
|   Saturday | Sunday -> return 0
end case
```

6.4.7 If statement

The "**if**" construct allows conditional computations; it is generally included in all languages. In Lotos NT it has the form:

```
if $E_0$ then $I_0$
 elsif $E_1$ then $I_1$
 ...
 elsif $E_n$ then $I_n$
 [else $I_{n+1}$]
end if
```

where $n \geq 0$, so the "**elsif**" clauses are optional. A missing "**else**" clause is equivalent to "**else null**".

The expressions $E_0, ..., E_n$ are called *conditions*. They must be of type `bool`, and do not have side effects. The statements $I_0, ..., I_{n+1}$ should return a value of the same type and should initialize the same variables. This constraint is important for the control of the variable flow.

The evaluation of an "**if**" statement is done as follows. The conditions $E_0, ..., E_n$ are evaluated in this order until a condition E_i evaluates to **true**; the evaluation of the "**if**" statement is the same as the evaluation of I_i. If all conditions are false, the result of "**if**" is the result of I_{n+1}, which by default is "**null**".

Example 6.4.2
The following statement computes the maximum of two integral numbers `X` and `Y`:

```
if X >= Y then return X else return Y end if
```

■

The "**if**" statement is less powerful than the "**case**" statement. Moreover, the "**if**" statement above may be translated into the following (equivalent) "**case**" statement:

```
case $E_0$ in
   true -> $I_0$
 | false ->
       case$E_1$ in
          true -> $I_1$
        | false -> ...
       end case
end case
```

However, in this case it is recommended to use the "**if**'" statement instead of the sophisticated "**case**" statement.

6.4.8 Iteration Statements

LOTOS NT allows to describe repetitive processing of data by mean of several iteration constructs. The most general and simplest one is the unbreakable loop, but several specialized iteration constructs are also provided, like breakable, conditional ("**while**"), and iterative ("**for**") loops.

Iterative constructs provide a way to express recursive processing of data without use of recursive functions. This avoids the stack overflow due to the infinite or great number of recursive function calls.

Loop forever statement The simplest iterative construct offered by LOTOS NT is the "**loop**" forever statement:

loop *I* **end loop**

where *I* is called the loop *body*.

The evaluation of a "**loop**" forever statement never terminates. The statement *I* is repeatedly evaluated. It can read and write variables of the current context. This type of iteration may introduce non-terminating processing of data. This may be signaled by the compiler. Note that if *I* raises a handled event (§ 6.4.9, p. 56), the loop is interrupted.

Breakable loop statements In practice, it is difficult to imagine examples of data processing which never terminate. Statements are generally used to compute (instantaneously) values. For this reason a form of breakable loop is provided:

loop *X* **in**
  I
end loop

where *X* is the loop identifier, *i.e.*, the loop name. The statement *I* may read and write variables of the current context with respect to static semantics constraints.

A loop is broken using the "**break**" statement which has the following syntax:

break *X*

where *X* is the name of the loop to be broken.

Example 6.4.3

The following statement computes the sum of elements of a given list of integers xs:

```
var ys: intlist := xs, total: int := 0
in
    ys := xs;
    total := 0;
    loop Sum in
        case ys
            var z: int, zs: intlist in
            nil -> break Sum
        |   cons (z, zs) -> total := total + z;
```

```
                  ys := zs
        end case
      end loop
    end var
```

■

The named loops are used to break loops which are not the inner one, for example:

```
    loop fred in
      loop janet in
        if V then break fred
        ...
```

As will be shown in Section 6.4.9, the breakable loop is a syntactic sugar for infinite loop construct and event handling.

While statement The conditional execution of a loop may be expressed using the "**while**" construct which exists in most languages:

```
while $E_0$ loop
  $I$
end while
```

where E_0 is an expression of type `bool`. I is the body of the loop, which may return a result (not used), read or write the variables of the current context.

At each iteration the expression E_0 is evaluated; if it returns **true**, the statement I is evaluated; otherwise, the "**while**" statement terminates.

Example 6.4.4
The statement below computes the factorial of `n`:

```
var k: int,
    fact: int
in
    k := n;
    fact := 1;
    while (k > 0) loop
        fact := fact * k;
        k := k - 1
    end while;
    return fact
end var
```

The property `fact` = `n`!/`k`! is the invariant of the loop, and the termination is ensured by the fact that `k` decreases at each iteration. The result of the statement is `fact` = `n`!/0! = `n`!. ■

This form of loop may be translated into a breakable loop as follows:

```
loop $X$ in
  if $E_0$ then $I$
  else break $X$
  end if
end loop
```

A breakable "**while**" loop of the form "**while** E_0 **loop** X **in** I **end loop**" is also available. Its semantics is the same as the unbreakable "**while**" loop, except that the loop is interrupted if I executes a "**break** X" statement, similarly to a breakable "**loop**".

For statement The last iterative construct, the "**for**" statement, allows to describe in a compact form finite iterations. Its form is closed to the "**for**" construct of the C language:

```
for $I_0$ while $E_1$ by $I_2$ loop
  $I$
end loop
```

where I_0 is a statement doing only variable assignments; E_1 is an expression of type `bool`; I_2 is a statement doing only assignments; I is the body of the loop, repeatedly executed. It can assign variable but it cannot return a result.

The evaluation of a "**for**" statement begins with evaluating the initialization statement I_0 in the current context of variables. Then, while the boolean expression E_1 evaluates to **true**, the body of the loop, I, is evaluated and when I terminates then I_2 is evaluated. If E_1 evaluates to **false**, the "**for**" statement terminates.

In fact, this form of loop is syntactic sugar of breakable loop; it can be translated as follows:

```
$I_0$;
loop $X$ in
  if $E_1$ then $I$; $I_2$
  else break $X$
  end if
end loop
```

A breakable "**for**" loop of the form "**for** I_0 **while** E_1 **by** I_2 **loop** X **in** I **end loop**" is also available. Its semantics is the same as the unbreakable "**for**" loop, except that the loop is interrupted if I executes a "**break** X" statement, similarly to a breakable "**loop**".

6.4.9 Events and their handling

An important feature of the language is the possibility to signal and treat the errors or unexpected cases by raising and trapping events (formerly also called *exceptions*).

Raise statement Events are used to interrupt the evaluation of expressions or statements. An event may be raised using the "**raise**" statement, whose simplest form is:

```
raise $X$
```

X is an event identifier which should be already declared. The declaration of an event is made using the "**trap**" statement.

The evaluation of a "**raise**" signals the event X and blocks the evaluation.

Example 6.4.5
The following specification raises the event `Hd` if one tries to take the head of an empty list:

```
case xs
    var x: int in
    nil -> raise Hd
```

```
|   cons (x, any) -> return x
end case
```

■

Assert statement Events may be raised by the violation of an assertion, using the "**assert**" statement, whose general form is:

assert E **raise** X

X is an event identifier which should be already declared and E is a boolean expression. If E is false then the "**assert**" statement behaves as "**raise** X". Otherwise, it behaves as "**null**".

The simpler form "**assert** E" is equivalent to "**assert** E **raise** `UNEXPECTED`".

Trap statement Events either propagate to top level, or are *trapped* by a "**trap**" construct containing the event handler. The simplest syntax of the "**trap**" construct is:

```
trap
  exception $X_1$ is $I_1$
  ...
  exception $X_n$ is $I_n$
in
  $I_0$
end trap
```

where $n \geq 0$. The clauses contained between keywords "**trap**" and "**in**" are called *event* (formerly also called *exception handler*). An event handler declares the name of the event X_i and its treatment I_i.

The scope of event identifiers $X_1, ..., X_n$ is only the statement I_0. So events $X_1, ..., X_n$ are handled only if raised by I_0. If one of $I_1, ..., I_n$ raises an event X_i, it is not handled by the current "**trap**" statement. The event identifiers $X_1, ..., X_n$ must be different from `UNEXPECTED` and `i`.

The statements $I_1, ..., I_n$ may either return a value of the same type and initialize the same variables as the statement I_0, or block. These constraints are checked statically. They ensure that the "**trap**" statement is well typed, and the flow of initialized variables is the same whatever the evolution of evaluating I_0 is.

The evaluation of the "**trap**" statement begins with the evaluation of I_0. If I_0 raises one of the events X_i, its evaluation is interrupted, and the result of the "**trap**" statement is the result of I_i. If I_0 terminates normally, *i.e.*, without raising any of the events X_i, the "**trap**" statement also terminates.

Example 6.4.6
The following statement returns 0 when `c` equals 0, or assigns the value of `b/c` to `a` otherwise.

```
trap
    exception ZD is return 0
in
    a := b /[ZD] c
end trap
```

■

The "**trap**" statement both declares and traps the event—this means it is impossible for an event to escape outside its scope, except the `UNEXPECTED` predefined event.

NOTE: This can be contrasted with a language such as SML where exception declaration and exception handling are separated, so it is possible for exceptions to escape their scope.

```
local
  exception Foo
in
  raise Foo
end
```

Note that the only way in which an event distinct from `UNEXPECTED` can be observed by its environment is by trapping it. The `UNEXPECTED` event stops the execution abruptly and prints a message giving the file and line where the event was raised. If the C code generated by Traian has been compiled with the `-DDEBUG` flag, then the names and locations of processes and functions that were on the call stack when the event was raised are also displayed, for debugging purpose.

Implementation issues If the code generated by Traian is used in external implementations, the C function `TRAIAN_INIT ()` must be called before any action. This function initializes the structures used by Traian for the implementation of the event mechanism.

6.4.10 Variable use

The "**use**" construct has the simple form:

use V_0 **,** ... **,** V_n

where $V_0, ..., V_n$ are variable identifiers. It marks the variables $V_0, \ldots, V_n$ as used, as if they occurred in some expression. This statement is useful to eliminate warnings signaling that some variables are unused.

The variables $V_0, \ldots, V_n$ must be distinct and declared in the environment.

6.4.11 Event access

The "**access**" construct has the simple form:

access X_0 **,** ... **,** X_n

where $X_0, ..., X_n$ are event identifiers. It marks the events $X_0, \ldots, X_n$ as accessed, as if they occurred in some expression or instruction. This statement is useful to eliminate warnings signaling that some events are unaccessed.

The events $X_0, \ldots, X_n$ must be distinct and declared in the environment.

6.5 Functions

Functions are a mean for code structuring and re-usability.

This section describes how Lotos NT users may define and use functions. The Lotos NT predefined functions are described in Section (§ 5.3, p. 31).

6.5.1 Function definition

A function definition has the following syntax:

> **function** $F\ [\ \texttt{[}\overline{X}_1\texttt{: none, ...,}\ \overline{X}_m\texttt{: none]}\ ]\ \texttt{(}[A_1]\ \overline{V}_1\texttt{:}T_1\texttt{, ..., }[A_n]\ \overline{V}_n\texttt{:}T_n\texttt{)}\ [\texttt{:}T]$ **is**
> $[\mathit{precondition}_1\,\texttt{;}\ \ldots\mathit{precondition}_p\,\texttt{;}]$
> $[\mathit{postcondition}_1\,\texttt{;}\ \ldots\mathit{postcondition}_q\,\texttt{;}]$
> I
> **end function**

where A_i may be "**in**", "**in var**", "**out**", "**out var**", or "**in out**". The default value for A_i is "**in**".

NOTE: This form for function declaration was chosen for syntactic compatibility with IDL (and Ada) and for an easier interface with C.

F is the name of the function. Two function names may be the same if their profiles (*i.e.*, the types of parameters or the result type) differ.

$\texttt{(}[A_i]\ \overline{V}_1\texttt{:}T_1\texttt{, ...)}$ is the list of *formal value parameters*. Value parameters may be constant values ("**in**" or "**in var**" parameter), result values ("**out**" or "**out var**" parameter), or modifiable values ("**in out**" parameters); the default type is "**in**". An "**in**" parameter may be read but its value is not changed by the function call. If its value is changed by I, then a warning is issued, unless it is declared as "**in var**", which allows it to be used as a local variable. An "**out**" parameter should be assigned by I and its value is visible after the function call. If its value is read by I (after being assigned), then a warning is issued, unless it is declared as "**out var**", which allows it to be used as a local variable. An "**in out**" parameter has an initial value, and I may modify them; the value of the parameter assigned by I is visible after the function call. The scope of variables in the lists $\overline{V}_1, ..., \overline{V}_n$ is the body of the function, I.

T is the result type of the function.

$\texttt{[}\overline{X}_1\texttt{: none, ...]}$ is the list of *formal event parameters* (formerly also called *formal exception parameters*). The scope of the events in the lists $\overline{X}_1, ..., \overline{X}_m$ is the body of the function, I. The events in the lists $\overline{X}_1, ..., \overline{X}_m$ must be different from `UNEXPECTED` and `i`.

Each *precondition* has the form "**require** E [**raise** X [`()`]]", where E is a boolean expression whose variables must be declared as "**in**", "**in var**", or "**in out**" parameters and X (if present) is either an event parameter or the `UNEXPECTED` event. The expression E must evaluate to true when entering the function. Otherwise, the exception X (if present) or `UNEXPECTED` (otherwise) is raised.

Each *postcondition* has the form "**ensure** E [**raise** X [`()`]]", where E is a boolean expression whose variables must be declared as parameters and X (if present) is either an event parameter or the `UNEXPECTED` event. The expression E must evaluate to true when exiting the function. Otherwise, the exception X (if present) or `UNEXPECTED` (otherwise) is raised.

In a postcondition, the value expression "V`.`**in**" can be used to denote the input value of V if V is a parameter declared with mode "**in**", "**in var**", or "**in out**". The value expression "V`.`**out**" can be used to denote the output value of V if V is a parameter declared with mode "**out**", "**out var**", or "**in out**". If an "**in out**" parameter V occurs in a postcondition, then it must be used either in the form "V`.`**in**" or "V`.`**out**". For this reason, the notation "**...**" cannot be used in a postcondition if its expansion contains a parameter V declared with mode "**in out**" in the current function. V may also contain occurrences of the keyword "**result**" to denote the function result, if any.

The statement I computes the result value of the function and the output parameters. Its environment is the list of formal parameters (value and event). In LOTOS NT it is not possible to assign "global" variables or to raise "global" events. All variables and events used by the body of the function must be declared as function parameters, with the exception of the special events `i` and `UNEXPECTED`. If T

is given, the result type of I must be T. The values assigned to output parameters must be correctly typed.

Example 6.5.1
Consider the declaration of the function `hd`:

```
function hd [Hd: none] (xs: intlist) : int
is
    case xs var x: int in
        nil -> raise Hd
    |   cons (x, any) -> return x
    end case
endfun
```

where `xs` is an input parameter of type `intlist`. Note that the body of the function does not assign global variables, the variable `x` being local to the second clause of the "**case**" statement.

The declaration below uses the output parameters to return several results:

```
function partition (in xs: intlist, out less: intlist, in x: int, out gtr: intlist)
is
    var ys: intlist,
        ls: intlist,
        gs: intlist
    in
        ys := xs;
        ls := nil;
        gs := nil;
        loop P in
            case ys var z: int, zs: intlist in
                nil -> break P
            |   cons (z, zs) ->
                    if (z < x) then
                        ls := cons (z, ls)
                    else
                        gs := cons (z, gs)
                    end if;
                    ys := zs
             end case
        end loop;
        less := ls;
        gtr  := gs
    end var
end function
```

■

Functions whose name is a special identifier are aimed at being used in value expressions only. Thus, they must have a result type and must not have parameters of mode "**out**", "**out var**", or "**in out**".

6.5.2 Function call

Function calls can be used in both expressions and statements. In expressions, functions that have two parameters can be used either in the prefix form or in the infix form.

The call of a function F in LOTOS NT has two forms. The "positional" function call give the ordered list of the parameters:

> [**eval**] [V:=] F [X'_1, ..., X'_m] ([E_1|?V'_1|!?V'_1], ..., [E_n|?V'_n|!?V'_n])

where n, *resp.* m, must be equal to the number of formal value parameters, *resp.* to the number of formal event parameters. ([?V'_1|E_1], ...) is the list of *actual value parameters*. Expressions E_i should appear in the same position as the "**in**" and "**in var**" parameters and must have the same type. Variables V_i should appear as actual parameters of the "**in out**" (when prefixed by "!?") and "**out**" or "**out var**" (when prefixed by "?") formal parameters, and must be already declared with the same type as the formal parameters. $X'_1, ..., X'_n$ are *actual event parameters*. The result of the function, if any, may be assigned to the variable V.

The "named" call of the functions use the name of formal parameters to specify the correspondence between formal and actual parameters; the order of the actual parameters is not important. The three alternatives below are equivalent:

> [**eval**] [V:=] F [X'_1, ..., X'_m] (V_1->[E_1|?V'_1|!?V'_1], ..., V_n->[E_n|?V'_n|!?V'_n])

where the list of actual value parameters is named,

> [**eval**] [V:=] F [X_1->X'_1, ..., X_m->X'_m] ([E_1|?V'_1|!?V'_1], ..., [E_n|?V'_n|!?V'_n])

where the list of actual event parameters is named, and

> [**eval**] [V:=] F [X_1->X'_1, ..., X_m->X'_m]
> (V_1->[E_1|?V'_1|!?V'_1], ..., V_n->[E_n|?V'_n|!?V'_n])

where both lists are named. The constraints above are also applied here.

Note that the **eval** keyword is always optional in functions.

Note that the "positional" style cannot be merged with the "named" style in the same list of actual parameters. This style of function call is similar to the Ada style.

The "**...**" shorthand for the record of actual parameters VS is expanded to the list of unspecified parameters as follows: if the parameter is an "**in**" or "**in var**" one, the expression is the variable which has the same name as the formal parameter; if the parameter is an output ("**in out**", "**out**", or "**out var**"), the actual parameter is the query symbol followed by the name of the formal parameter. Similarly for the list of actual event parameters.

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

Note that LOTOS NT supports call by value ("**in**" and "**in var**" parameters) and (a restricted form of) call by reference ("**in out**", "**out**", and "**out var**" parameters). Functions as arguments of functions (second order functions) are not allowed.

Example 6.5.2
The function `partition` may be used by a quick sort function as follows:

```
function quicksort (xs: intlist) : intlist
is
    case xs var y: int, ys: intlist in
        nil -> nil
    |   cons (y, ys) ->
            var l: intlist, g: intlist
            in
                partition (xs, ?l, y, ?g);
                return append (quicksort (l), cons (y, quicksort (g)))
            end var
     end case
end function
```

Note that the variables `l` and `g` are locally declared to the second clause of the case because the first clause does not initialize them. ■

Chapter 7

Channels, Behaviours, and Processes

7.1 Channels

A channel denotes a type constraint over the communication offers that can be used by events in rendezvous. A channel name is denoted by the letter H.

The syntax for channel definition is the following:

> **channel** H **is**
> $\;\;$ ([$\overline{V}:T$ { , $\overline{V}:T$ }])
> $\;\;$ { , ([$\overline{V}:T$ { , $\overline{V}:T$ }]) }
> **end channel**

If a channel H contains a declaration of the form "($V_1:T_1$, ..., $V_n:T_n$)", then every event typed with H can be used in a rendezvous with n offers typed respectively by $T_1, \ldots, T_n$.

Beyond such user-defined channels, there are two special channels:

- The keyword **any** denotes a built-in channel that defines no constraint on events. Thus, any event declared with channel **any** can be used in a rendezvous involving any number and type of offers.

- The channel **none** is defined by "**channel none is** () **end channel**" in the predefined library. Thus, any event declared with channel **none** can be used only in rendezvous involving no offer at all.

Note that user-defined channels are recognized by the parser, but are not yet fully implemented.

7.2 Behaviours

The following grammar gives the syntax of behaviours. Behaviours followed by a star (*) are not yet fully implemented.

B	::=	**null**	*termination*	(B 1)
	|	**stop**	*deadlock*	(B 2)
	|	V := E	*assignment**	(B 3)
	|	V [E] := E	*array assignment**	(B 4)
	|	V := **any** [T][**where** E]	*nondeterministic assignment**	(B 5)
	|	B ; B	*sequential composition*	(B 6)
	|	**var** $\overline{V}$:T {,$\overline{V}$:T} **in**	*variable declaration**	(B 7)
		B		(B 8)
		end var		(B 9)
	|	**case** E {,E}	*case behaviour**	(B 10)
		[**var** $\overline{V}$:T {,$\overline{V}$:T}] **in**		(B 11)
		BM		(B 12)
		end case		(B 13)
	|	**if** E **then** B	*conditional behaviour**	(B 14)
		{ **elsif** E **then** B }		(B 15)
		[**else** B]		(B 16)
		end if		(B 17)
	|	**only if** E **then** B	*only if behaviour**	(B 18)
		{ **elsif** E **then** B }		(B 19)
		end if		(B 20)
	|	**select** B { [] B } **end select**	*select behaviour**	(B 21)
	|	[**eval**] V := F [[XS]] [(VS)]	*procedure call with result**	(B 22)
	|	**eval** F [[XS]] [(VS)]	*procedure call without result**	(B 23)
	|	P [[XS]] [(VS)]	*process call**	(B 24)
	|	**breakable loop** B **end loop**	*forever loop**	(B 25)
	|	**loop** X **in** B **end loop**	*breakable loop**	(B 26)
	|	**while** E **loop** B **end loop**	*while loop**	(B 27)
	|	**while** E **loop** X **in** B **end loop**	*breakable while loop**	(B 28)
	|	**for** B **while** E **by** B **loop** B **end loop**	*for loop**	(B 29)
	|	**for** B **while** E **by** B **loop** X **in** B **end loop**	*breakable for loop**	(B 30)
	|	**break** X	*loop break**	(B 31)
	|	**raise** X [()]	*raise event*	(B 32)
	|	**assert** E [**raise** X [()]]	*assertion*	(B 33)
	|	**trap** {**exception** X **is** B} **in**	*trapping events**	(B 34)
		B		(B 35)
		end trap		(B 36)
	|	**par** [XS **in**]	*parallel composition*	(B 37)
		[XS ->] B		(B 38)
		{ ||[XS ->] B }		(B 39)

		end par		(B 40)
	|	**hide** $\overline{X}$:H{, $\overline{X}$:H} **in** B **end hide**	*event hiding**	(B 41)
	|	**disrupt** B **by** B **end disrupt**	*disrupt**	(B 42)
	|	X [(OS)][**where** E]	*rendezvous**	(B 43)
	|	**use** V { , V }	*variable use**	(B 44)
	|	**access** X { , X }	*event access**	(B 45)

BM	::=	P {,P} {| P {,P} -> B	*match-behaviour*	(BM1)
	|	BM | BM	*list*	(BM2)

OS	::=	O_0, ..., O_n	*positional style*	(OS 1)
	|	V_0 -> O_0, ..., V_n -> O_n [,...]	*named style*	(OS 2)

O	::=	E	*send offer*	(O 1)
	|	?P	*receive offer*	(O 2)

In the following we present the Lotos NT behaviours when they are implemented.

7.2.1 Stop Behaviour

The behaviour "**stop**" blocks the execution: no further rendezvous is possible. This behaviour never terminates: it represents a deadlock.

7.2.2 Rendezvous

So far, rendezvous is restricted to the form "X (E)". Its execution performs a rendezvous on event X with value expression E. Concretely, this currently amounts to print to the standard output a line "X' !E'", where X' is the event X with all lower case letters converted to upper case, and where E' is the result of evaluating expression E. This syntax follows the conventions of transition labels in LNT.

7.2.3 Sequential Composition

The execution of the sequential composition of behaviours "B_1 ; B_2" starts by executing B_1. Only if B_1 terminates successfully, B_2 is executed.

7.2.4 Process Call

Most process calls have the form "P `[`XS`]`" or "P `[`XS`] (`VS`)`". However, calls to processes without event parameters have the simpler form "P" or "P `(`VS`)`", where brackets do not appear. If P is also the name of an event and arguments VS have neither form "`!?`V" nor "V' `-> !?`V", then this behaviour may either denote a call to process P or a rendezvous. In that case, TRAIAN considers the behaviour as a rendezvous and issues a warning. To avoid the warning, either the process or the event has to be renamed.

7.2.5 Function call

In behaviours, to avoid confusion between function call, process call, and rendezvous, the **eval** keyword (which is always optional in statements) is mandatory when calling a function that does not return any result. The **eval** keyword remains optional when calling a function that returns a result, which is necessarily assigned to a variable.

Example 7.2.1
The **eval** keyword is mandatory in the following behaviour (where F is a function):

```
hide G : any in var Y : nat in
      eval F (1, ?Y);
      G (Y)
end var end hide
```

■

Example 7.2.2
The **eval** keyword is mandatory in the following behaviour, i.e., the following behaviours are equivalent:

```
hide G : any in var Y, Z : nat in
      eval Y := F (1, ?Z);
      G (Y, Z)
end var end hide
```

and

```
hide G : any in var Y, Z : nat in
      Y := F (1, ?Z);
      G (Y, Z)
end var end hide
```

■

7.3 Process Definition

A process definition has the following syntax:

```
process $P$ [[$\overline{X_1}$:$H_1$, ..., $\overline{X_m}$:$H_m$]] [([$A_1$] $\overline{V}_1$:$T_1$, ..., [$A_n$] $\overline{V}_n$:$T_n$)] is
   [$precondition_1$; ... $precondition_p$;]
   [$postcondition_1$; ... $postcondition_q$;]
   $B$
end process
```

The current version of LOTOS NT supports only a single process (called principal process) without value parameters. The behaviour of the principal process is restricted to a sequence of rendezvous with a single send offer. This process is executed and allows to display the results of a sequence of expression evaluations.

Example 7.3.1
The following process

```
process MAIN [PRINT: any] is
   print (”text”);
   Print (1.0);
   PRINT (FACTORIAL (2))
end process
```

prints to the standard output the following three lines

```
"PRINT !text"
"PRINT !1"
"PRINT !2"
```

■

Appendix A

Syntax Summary

This chapter presents the full concrete grammar (syntax) of the language. The notations used are those presented in Chapter 2. The lexical structure of the language is defined in Chapter 3. The entry point of the grammar is the nonterminal symbol *descr*.

A.1 Syntax of the module part

Identifiers:

C	*constructor identifier*	(id1)
F	*function (non-constructor) identifier*	(id2)
H	*channel identifier*	(id3)
K	*constant identifier*	(id4)
mod-id	*module identifier*	(id5)
P	*process identifier*	(id6)
T	*type identifier*	(id7)
V	*variable identifier*	(id8)
X	*event identifier*	(id9)

Module body:

MB	::=	$D\ \{D\}$	*declarations*	(MB1)

Unit declaration:

UD	::=	**module** *mod-id*$_0$ [`(`*mod-id*$_1$`,` ... `,` *mod-id*$_n$`)`]	*simple module*	(UD1)
		[**with** F_0`,` ... `,`F_n] **is**		
		module_pragma$_1$ ... *module_pragma*$_n$		
		MB		

end module

Module pragma:

module_pragma ::= **!int_bits** `INTEGER` (module_pragma1)
| **!int_check** (`0` | `1`)
| **!int_inf** [`+` | `-`] `INTEGER`
| **!int_sup** [`+` | `-`] `INTEGER`
| **!nat_bits** `INTEGER`
| **!nat_check** (`0` | `1`)
| **!nat_inf** `INTEGER`
| **!nat_sup** `INTEGER`
| **!num_bits** `INTEGER`
| **!num_card** `INTEGER`
| **!string_card** `INTEGER`
| **!update** `STRING`
| **!version** `STRING`

Description:

descr ::= *UD* LOTOS NT *description* (descr1)

Declarations:

D ::= **type** T **is** *type* (D1)
[**!external**]
[**!implementedby** `STRING`]
[**!comparedby** `STRING`]
[**!printedby** `STRING`]
[**!iteratedby** `STRING, STRING`]
[**!pointer**]
[**!nopointer**]
[**!bits** [`INTEGER`]]
[**!card** [`INTEGER`]]
[**!list**]
TD
[**with** F {,F}]
end type
| **function** F [[XL]] [(VFL)] [:T] **is** *function* (D2)

[$precondition_1$; . . . $precondition_p$;]
[$postcondition_1$; . . . $postcondition_q$;]
I
end function
| **channel** H **is** *channel* (D3)
([$\overline{V}$:T { , $\overline{V}$:T }])
{ , ([$\overline{V}$:T { , $\overline{V}$:T }]) }
end channel
| **process** P [[XL]] [(VFL)] **is** *main process* (D4)
[$precondition_1$; . . . $precondition_p$;]
[$postcondition_1$; . . . $postcondition_q$;]
B
end process

A.2 Syntax of the data part

Attributes of parameters:

A ::= [**in** [**var**]] *input formal parameter* (A1)
| **out** [**var**] *output formal parameter* (A2)
| **in out** *input/output formal parameter* (A3)

List of variables:

$\overline{V}$ ::= V {,V} *list of variable identifiers* (VL1)
VL ::= $\overline{V}$:T {,$\overline{V}$:T} *list of variables* (VL2)

VFL ::= $A\,\overline{V}$:T {,$A\,\overline{V}$:T} *formal parameter list* (VFL1)

List of events:

$\overline{X}$ ::= X {,X} *list of event identifiers* (XL1)
XL ::= $\overline{X}$: H {,$\overline{X}$: H} *list of events* (XL2)

Precondition:

precondition ::= **require** E [**raise** X [()]] (req1)

Postcondition:

postcondition	::=	**ensure** E [**raise** X [`()`]]	(ens1)

Type definition:

array_bound	::=	`INTEGER`	*unsigned integer*	(array_bound1)

range_bound	::=	`INTEGER`	*unsigned integer*	(range_bound1)
	$\mid$	$(\texttt{+} \mid \texttt{-})$ `INTEGER`	*signed integer*	
	$\mid$	`CHAR`	*character*	

TD	::=	$C\ [\texttt{(}VL\texttt{)}]\ \{,\ C\ [\texttt{(}VL\texttt{)}]\}$	*constructed type*	(TD1)
	$\mid$	**list of** T	*list*	(TD2)
	$\mid$	**sorted list of** T	*sorted list**	(TD3)
	$\mid$	**set of** T	*set**	(TD4)
	$\mid$	**sorted set of** T	*sorted set**	(TD5)
	$\mid$	**array** `[`*array_bound*`..`*array_bound*`]` **of** T	*array**	(TD6)
	$\mid$	**range** *range_bound*`..`*range_bound* **of** T	*range**	(TD7)
	$\mid$	X:T **where** V	*predicate type**	(TD8)

Sequence of expressions:

ES	::=	$[\ldots]$	*empty or wildcard*	(ES1)
	$\mid$	$V \Rightarrow E\ \{,V \Rightarrow E\}\ [,\ldots]$	*disjoint union*	(ES2)
	$\mid$	$E\ \{,E\}$	*list*	(ES3)
UES	::=	$V \Rightarrow E\ \{,V \Rightarrow E\}\ [,\ldots]$	*disjoint union*	(ES4)

Sequence of events:

XS	::=	$[\ldots]$	*empty or wildcard*	(XS1)
	$\mid$	$X \Rightarrow X\ \{,X \Rightarrow X\}\ [,\ldots]$	*disjoint union*	(XS2)
	$\mid$	$X\ \{,X\}$	*list*	(XS3)

Expressions:

E	::=	K	*primitive constant*	(E1)

	$\mid$	V	*variable*	(E2)
	$\mid$	V **.in**	*input argument (in postcondition)*	(E3)
	$\mid$	V **.out**	*output argument (in postcondition)*	(E4)
	$\mid$	**result**	*function result (in postcondition)*	(E5)
	$\mid$	$C\ [\texttt{(}ES\texttt{)}]$	*constructor application*	(E6)
	$\mid$	$E\ C\ E$	*infix constructor application*	(E7)
	$\mid$	$'\{'\ [E\ \{\texttt{,}E\}]\ '\}'$	*list or set construction*	(E8)
	$\mid$	$F\ [\texttt{[}XS\texttt{]}]\ [\texttt{(}ES\texttt{)}]$	*function call*	(E9)
	$\mid$	$E\ F\ [\texttt{[}XS\texttt{]}]E$	*infix function call*	(E10)
	$\mid$	$E\texttt{.}[\texttt{[}X\texttt{]}]V$	*field selection*	(E11)
	$\mid$	$E\texttt{.}[\texttt{[}X\texttt{]}]'\{'\ UES\ '\}'$	*field update*	(E12)
	$\mid$	E **of** T	*type coercion*	(E13)
	$\mid$	$\texttt{(}E\texttt{)}$	*parenthesized expression*	(E14)

The precedence of operators appearing in expressions is given on table A.2.

Priority	*Operations*
0.	**of**, `.` field selection and update
1.	infix operators not listed below
2.	`**`
3.	`*`, `/`, `div`, `mod`, `rem`
4.	`+`, `-`
5.	`==`, `=`, `!=`, `<`, `<=`, `>=`, `>`
6.	`and`, `and then`, `or`, `or else`, `xor`, `=>`, `<=>`

Sequences of patterns:

PS	$::=$	$[\texttt{...}]$	*empty or wildcard*	(PS1)
	$\mid$	$V \Rightarrow P\ \{\texttt{,}V \Rightarrow P\}\ [\texttt{,...}]$	*disjoint union*	(PS2)
	$\mid$	$P\ \{\texttt{,}P\}$	*list*	(PS3)

Patterns:

P	$::=$	V	*variable*	(P1)
	$\mid$	**any** $[T]$	*wildcard*	(P2)
	$\mid$	V **as** P	*aliasing*	(P3)
	$\mid$	K	*constant*	(P4)
	$\mid$	$C\ [\texttt{(}PS\texttt{)}]$	*constructed pattern*	(P5)
	$\mid$	$P\ C\ P$	*constructed pattern infixed*	(P6)
	$\mid$	$'\{'\ [P\ \{\texttt{,}P\}]\ '\}'$	*list pattern*	(P7)
	$\mid$	P **of** T	*explicit typing*	(P8)
	$\mid$	P **where** E	*guarded pattern*	(P9)
	$\mid$	$\texttt{(}P\texttt{)}$	*parenthesized pattern*	(P10)

Infix constructors in patterns obey the same precedence rules as in expressions (see table A.2), except "`and then`" and "`or else`", which are not permitted.

Match statements:

IM	::=	$P\ \{\texttt{,}P\}\ \{\texttt{\|}\ P\ \{\texttt{,}P\}\}$ `->` I	*match-statement*	(IM1)
	|	IM `|` IM	*list*	(IM2)

Actual value parameters:

VE	::=	E	*actual parameter "in" or "in var"*	(VE1)
	|	`?`V	*actual parameter "out" or "out var"*	(VE2)
	|	`!?`V	*actual parameter "in out"*	(VE3)

VS	::=	[**`...`**]	*empty or wildcard*	(VS1)
	|	$V \Rightarrow VE\ \{\texttt{,}V \Rightarrow VE\}\ [\texttt{,}$**`...`**$]$	*disjoint union*	(VS2)
	|	$VE\ \{\texttt{,}VE\}$	*list*	(VS3)

Statements:

I	::=	**return** E	*value return*	(I 1)
	|	**null**	*termination*	(I 2)
	|	V`:=`E	*assignment*	(I 3)
	|	I `;` I	*sequential*	(I 4)
	|	**var** $\overline{V}\texttt{:}T\ \{\texttt{,}\overline{V}\texttt{:}T\}$ **in** I **end var**	*variable declaration*	(I 5)

	|	**case** $E\ \{\texttt{,}E\}$ [**var** VL] **in**
		IM
		end case
			case statement	(I 6)

	|	**if** E **then** I
		{ **elsif** E **then** I }
		[**else** I]
		end if
			conditional statement	(I 7)

	|	[**eval**] [V`:=`] F [`[`XS`]`] [`(`VS`)`]	*procedure call*	(I 8)
	|	**loop** I **end loop**	*forever loop*	(I 9)
	|	**loop** X **in** I	*breakable loop*	(I 10)

	end loop		
|	**while** E **loop**	*while loop*	(I 11)
	I		
	end loop		
|	**while** E **loop** X **in**	*breakable while loop*	(I 12)
	I		
	end loop		
|	**for** I **while** E **by** I **loop**	*for loop*	(I 13)
	I		
	end loop		
|	**for** I **while** E **by** I **loop** X **in**	*breakable for loop*	(I 14)
	I		
	end loop		
|	**break** X	*break loop*	(I 15)
|	**raise** X [`()`]	*raise event*	(I 16)
|	**assert** E [**raise** X [`()`]]	*assertion*	(I 17)
|	**trap** {**exception** X **is** I} **in**	*trapping events*	(I 18)
	I		
	end trap		
|	**use** V {,V}	*variable use*	(I 19)
|	**access** X {,X}	*event access*	(I 20)

A.3 Syntax of the behaviour part

Offers:

O	::=	[V `->`][`!`]E	*send offer*	(O 1)
	|	[V `->`] `?`P	*receive offer*	(O 2)

Sequence of offers:

OS	::=	O_0 `,` ..., `,` O_n	*positional style*	(OS 1)
	|	V_0`->`O_0 `,` ..., `,` V_n`->`O_n [`,...`]	*named style*	(OS 2)

Match behaviours:

BM	::=	P {`,`P} {`|` P {`,`P}} `->` B	*match-behaviour*	(BM 1)
	|	BM `|` BM	*list*	(BM 2)

Behaviours:

```
B  ::= null                                         termination                      (B 1)
     | stop                                         deadlock                         (B 2)
     | V:=E                                         assignment*                      (B 3)
     | V:= any [ T ][ where E ]                     nondeterministic assignment*     (B 4)
     | B ; B                                        sequential composition           (B 5)
     | var V̄:T {,V̄:T} in                            variable declaration*            (B 6)
         B                                                                           (B 7)
       end var                                                                       (B 8)
     | case E {T,E}                                 case behaviour*                  (B 9)
       [ var V̄:T {,V̄:T} in                                                           (B 10)
         BM                                                                          (B 11)
       end case                                                                      (B 12)
     | if E then B                                  conditional behaviour*           (B 13)
       { elsif E then B }                                                            (B 14)
       [ else B ]                                                                    (B 15)
       end if                                                                        (B 16)
     | only if E then B                             only if behaviour*               (B 17)
       { elsif E then B }                                                            (B 18)
       end if                                                                        (B 19)
     | select B { [] B } end select                 select behaviour*                (B 20)
     | [eval] V:= F [[XS]] [(VS)]                   procedure call with result*      (B 21)
     | eval F [[XS]] [(VS)]                         procedure call without result*   (B 22)
     | P [[XS]] [(VS)]                              process call*                    (B 23)
     | loop B end loop                              forever loop*                    (B 24)
     | loop X in B end loop                         breakable loop*                  (B 25)
     | while E loop B end loop                      while loop*                      (B 26)
     | while E loop X in B end loop                 breakable while loop*            (B 27)
     | for B while E by B loop B end loop           for loop*                        (B 28)
     | for B while E by B loop X in B end loop breakable for loop*                   (B 29)
     | break X                                      loop break*                      (B 30)
     | raise X [()]                                 raise event                      (B 31)
     | assert E [raise X [()]]                      assertion                        (B 32)
     | trap {exception X is B} in                   trapping events*                 (B 33)
         B                                                                           (B 34)
       end trap                                                                      (B 35)
     | par [ XS in ]                                parallel composition             (B 36)
         [ XS -> ] B                                                                 (B 37)
         { ||[ XS -> ] B }                                                           (B 38)
       end par                                                                       (B 39)
```

$\mid$	**hide** $\overline{X}{:}H\{$**,** $\overline{X}{:}H\}$ **in** B **end hide**	*event hiding**	(B 40)
$\mid$	**disrupt** B **by** B **end disrupt**	*disrupt**	(B 41)
$\mid$	X [(OS)][**where** E]	*rendezvous**	(B 42)
$\mid$	**use** V { **,** V }	*variable use**	(B 43)
$\mid$	**access** X { **,** X }	*event access**	(B 44)

Bibliography

[BM79] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, New York, NY, 1979.

[Bol90] T. Bolognesi. A Graphical Composition Theorem for Networks of Lotos Processes. In IEEE Computer Society, editor, *Proceedings of the 10th International Conference on Distributed Computing Systems, Washington, USA*, pages 88–95. IEEE, May 1990.

[Cd95] J.P. Courtiat and R.C. de Oliveira. A Reachability Analysis of RT-LOTOS Specifications. Technical Report 95159, LAAS, May 1995.

[CGM$^+$96] Ghassan Chehaibar, Hubert Garavel, Laurent Mounier, Nadia Tawbi, and Ferruccio Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with 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 435–450. Chapman & Hall, October 1996. Full version available as INRIA Research Report RR-2958.

[dMRV92] Jan de Meer, Rudolf Roth, and Son Vuong. Introduction to Algebraic Specifications Based on the Language ACT ONE. *Computer Networks and ISDN Systems*, 23(5):363–392, 1992.

[GH93] Hubert Garavel and René-Pierre Hautbois. An Experiment with the Formal Description in LOTOS of the Airbus A340 Flight Warning Computer. In Maurice Nivat, Charles Rattray, Teodor Rus, and Giuseppe Scollo, editors, *First AMAST International Workshop on Real-Time Systems, Iowa City, Iowa, USA*, November 1993.

[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.

[Gut77] J. Guttag. Abstract Data Types and the Development of Data Structures. *Communications of the ACM*, 20(6):396–404, June 1977.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[ISO89a] ISO/IEC. LOTOS Description of the Session Protocol. Technical Report 9572, International Organization for Standardization – Open Systems Interconnection, Geneva, 1989.

[ISO89b] ISO/IEC. LOTOS Description of the Session Service. Technical Report 9571, International Organization for Standardization – Open Systems Interconnection, Geneva, 1989.

[ISO92a] ISO/IEC. Distributed Transaction Processing – Part 3: Protocol Specification. International Standard 10026-3, International Organization for Standardization – Information Technology – Open Systems Interconnection, Geneva, 1992.

[ISO92b] ISO/IEC. Formal Description of ISO 8072 in LOTOS. Technical Report 10023, International Organization for Standardization – Telecommunications and Information Exchange between Systems, Geneva, 1992.

[ISO92c] ISO/IEC. Formal Description of ISO 8073 (Classes 0, 1, 2, 3) in LOTOS. Technical Report 10024, International Organization for Standardization – Telecommunications and Information Exchange between Systems, Geneva, 1992.

[ISO95a] ISO/IEC. LOTOS Description of the CCR Protocol. Technical Report 11590, International Organization for Standardization – Open Systems Interconnection, Geneva, 1995.

[ISO95b] ISO/IEC. LOTOS Description of the CCR Service. Technical Report 11589, International Organization for Standardization – Open Systems Interconnection, Geneva, 1995.

[LL95] R. Lai and A. Lo. An Analysis of the ISO FTAM Basic File Protocol Specified in LOTOS. *Australian Computer Journal*, 27(1):1–7, February 1995.

[LL97] Luc Léonard and Guy Leduc. An Introduction to ET-LOTOS for the Description of Time-Sensitive Systems. *Computer Networks and ISDN Systems*, 29(3):271–292, 1997.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mun91] Harold B. Munster. LOTOS Specification of the MAA Standard, with an Evaluation of LOTOS. NPL Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[Pec94] Charles Pecheur. A proposal for data types for E-LOTOS. Technical Report, University of Liège, October 1994. Annex H of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS.

[QA92] J. Quemada and A. Azcorra. Structuring Protocols with Exception in a LOTOS Extension. In *Proceedings of the 12th IFIP International Workshop on Protocol Specification, Testing and Verification (Orlando, FL, USA).* North-Holland, June 1992.

[Sch88] Philippe Schnoebelen. Refined Compilation of Pattern-Matching for Functional Languages. *Science of Computer Programming*, 11:133–159, 1988.

[Sig99] Mihaela Sighireanu. *Contribution à la définition et à l'implémentation du langage "Extended LOTOS".* PhD thesis, Université Joseph Fourier (Grenoble), January 1999.

[WWF87] D. Watt, B. Wichmann, and W. Findlay. *ADA Language and Methodology.* Prentice-Hall, 1987.

Index

array, 37
break
    statement, 54
case
    statement, 52
else, 53
elsif, 53
exception, 57
false, 30, 32
for
    statement, 56
function, 59
if
    statement, 53
in out, 59
in var, 59
in, 51, 52, 57, 59
is, 59, 67
list, 36
loop
    breakable, 54–56
    forever, 54
    statement, 54
null, 51
of, 46
out var, 59
out, 59
process, 67
raise
    statement, 56
range, 38
set, 37
sorted list, 37
sorted set, 37
stop, 65
then, 53
trap
    statement, 57
true, 30, 32
var
    statement, 51
where, 38
while
    statement, 55
`TRAIAN_INIT`, 58

assignment
    statement, 51

BNF (Backus-Naum Form), 16

call
    by name, 44
    by position, 44
conditional
    statement, 52
constructor
    application, 44

data
    carrier, 17
    domain, 17

event
    raising, 56
exception
    handler, 57
    raising, 56
expression, 42
    parenthesized, 46

function, 58
    call, 61
        named, 61
        positional, 61
    definition, 59

handler, 57

iterative
    statement, 54

parameter

in out, 59, 61
in var, 59, 61
in, 59, 61
out var, 59, 61
out, 59, 61
actual, 61
event, 59
formal, 59
value, 59
pattern, 47
in case, 52
pragma, 38
pragmas
operation, 39
type, 38
process
definition, 67
main, 67

rendezvous, 65
return
statement, 51

selection, 45
sequential
behaviour, 65
statement, 51
statement, 49
syntax
concrete, 17

token
`BINARY_NUMBER`, 23
`CHAR`, 25
`DECIMAL_NUMBER`, 23
`HEX_NUMBER`, 23
`IDENTIFIER`, 21
`OCTAL_NUMBER`, 23
`REAL`, 24
`SPECIAL_IDENTIFIER1`, 22
`SPECIAL_IDENTIFIER2`, 22
`STRING`, 25
type, 29
array, 37
bool, 32
char, 33
constructor, 29
declaration, 29
enumerable, 35
enumerated, 35
finite, 35
finite enumerable (see scalar), 35
int, 32
list, 36
nat, 32
predicate type, 38
range, 37
real, 33
record, 35
scalar, 35
set, 37
sorted list, 36
sorted set, 37
string, 34
typing
explicit, 46

update, 46

value, 44
primitive constants, 41
variable, 44
declaration, 51