Guide Language Manual (V1.5)

Hiep NGUYEN VAN†, Michel RIVEILL†, Cecile ROISIN†
Richard COOPER‡and Francis WAI‡

April 91

†Unite Mixte Bull - IMAG
Z.I de Mayencin - 2, avenue de Vignate
F-38610 Gieres - FRANCE
e-mail : (riveill, roisin)@imag.imag.fr

‡Glasgow University
Department Computing Science
17, Lilybank Gardens
Glasgow G12 8Q2 - U.K

Abstract

This report is the Guide user manual, a language for programming distributed applications (version 1.5, december 1990). This language is closely integrated with the Guide distributed operating system, which provides its run-time support. Each feature of the language is described by its syntactic form and is illustrated by examples. The syntax chart, the base types, the type and class library, and the manual of the compiler are presented in appendices. A companion report presents illustrative examples of programs written in Guide.

NB : Comments should be sent to Michel Riveill or Cecile Roisin. Please point out in particular any discrepancy between this manual and the compiler.

1 Introduction

The programmer will find in this manual a precise description of the Guide language version 1.5 available on 15 december 1990. Its different constructs are shown in examples (all compiled and executed on the Guide system) and their definition is accompanied by the syntax rules necessary for their use. The syntax of the language is recapitulated in Appendix A. Appendix B contains the definition and rules of the base types of the language and of the predefined types and constructors ; the source code of the types in the Guide library is given in Appendix C; and Appendix D describes the operating principle of the compiler as well as its use. For a description of the object model of Guide and the justifications for this language, the reader should consult [1].

Conventions used to denote the syntax:

- terminals are written in capital letters or enclosed in angle brackets,

- non-terminals are enclosed in < and >,
- any number of x is written { x },
- at least one occurrence of x is written { x }*,
- at most one occurrence of x is written [x],
- exactly one occurrence of x is written (x),
- alternatives are denoted by |.

Examples in this manual use the following conventions :

- key words of the language are in capital letters,
- type, class and method identifiers have the first letter capitalised,
- other identifiers are entirely in lower case.

2 Programming Rules

2.1 Character Set

All the characters of the IA5 alphabet are allowed. The code '^L' (control key + L) may be used to cause a page skip during printing and is ignored by the compiler.

2.2 Identifiers

Identifiers must be alphanumeric (the first symbol being a letter) and may contain the underscore '_'. Character strings corresponding to the key words and reserved words must not be used. The language is case-sensitive. For example, 'MyIdent' and 'myident' are two different identifiers.

Type and class identifiers form two independent name spaces. It is thus possible to use the same identifier to designate a type and a class. This is often done when a type is implemented by a single class.

Syntax

<ident> = <letter> { <letter> | <digit> | '_' }
<letter> = ('A'..'Z' | 'a'..'z')
<digit> = ('0'..'9');

Example

```
Count, PERSON_1 and my_name  are correct identifiers ;
string?, 12age and _name are incorrect identifiers.
```

2.3 Key Words

```
ALL                         NOT
AND                         OF
ASSERTYPE                   OR
ATTRIBUTE                   OTHERS
BEGIN                       OUT
BY                          POWER
BLOCK                       PROCEDURE
CASE                        RAISE
CLASS                       READER
CONTROL                     REF
CO_BEGIN                    REPLACE
CO_END                      REPEAT
CONST                       RETRY
CONSTRUCTOR                 RETURN
DIV                         SELF
DO                          SIGNALS
ELSE                        SUBCLASS
END                         SUBTYPE
EXCEPT                      SUPER
EXCLUSIVE                   SYNONYM
FIRST                       SYSTEM
FOR                         THEN
FROM                        TO
IF                          TYPE
IMPLEMENTS                  TYPECASE
IN                          UNTIL
IN_OUT                      UNMOVABLE
IS                          WHILE
MOD                         WRITER
METHOD                      XOR
```

Key words may be written either completely in lower case or completely in upper case.

2.4 Symbols

```
=                           +
#  (difference)             -
>                           /
<                           |
<=                          *
>=                          &
```

2.5 Reserved Words

Types

```
Array[n] OF ...          LongInt
Boolean                  ShortInt
Char                     Record ... END
Integer                  String[n]
```

Constants

```
FALSE (or false)
TRUE (or true)
NIL (or nil)
```

Predefined environment variables

```
input
output
catal
myActivity
myDomain
```

Prohibited Words

To facilitate the use of future versions of the Guide compiler, the list of words which must not be used as identifiers in Guide programs is given below.

```
ADDRESS                  BEGIN_C
ATOMIC                   END_C
BEGIN_TRANSACTION        BEGIN_LOCK
END_TRANSACTION          END_LOCK
ABORT_TRANSACTION        INHERIT
                         OPERATOR
```

2.6 Comments

Single line comments are introduced by the string '//' and ended by the end-of-line character. Longer comments are now permitted and are put between "/*" and "*/".

Example

```
toto.Print ;  // here is a comment
// and another
// comment
/* here is a long comment. It may go on for several lines.
   These comments are not overlapping */
```

3 Guide Program

A Guide program is composed of a set of types and classes. For more information on the Guide object model, see reference [1].

The type of an object defines its behaviour by specifying the methods applicable to it. The syntax of this specification corresponds to the set of visible variables and

method signatures (or operations) which gives access to the objects of this type. It is also called the access interface. The semantics of a type corresponds to the formal (sic) specification of the effect of these methods on the object, given in the form of comments.

A class decribes one possible implementation of a type. A type may be implemented by several classes. A class defines: a data model (i.e. the set of variables which defines the state of the objects in that class); the code of the methods declared in the type which it is implementing; and, for each of these methods, an activation condition.

There are two sets of types and classes :

1. base types of the language decribed in Appendix B. They are divided into two categories :
   - simple types : Boolean, Integer, LongInt, ShortInt, and Char (cf. Appendix B 1) ;
   - structured types : Array [n] OF, Record, String[n] (cf. Appendix B 2) ;
2. Constructed types and classes. A set of standard constructed types and classes is given in the Guide library. These types are decribed in Appendix C.

Syntax

<type> = <base_type> | <constructed_type>
<base_type> = <elementary_type> | structured_type>
<elementary_type> = 'Integer' | 'ShortInt' | 'LongInt' | 'Char' | 'Boolean'
<structured_type> = record_type | array_type | string_type

Guide is a persistent object system, i.e. all the objects recognised by the Object Management System(OMS) are permanent and their destruction must be explicit. It may be carried out by the programmer (using method Destroy defined in the type Top - see Appendix C) or by the garbage-collector [3]. Persistence is made possible by the global reference system (or OID) of the objects managed by the system.

4 Objects, variables and references

4.1 Definitions

An object, the instance of a class, is a typed entity composed of a state (the data), and methods (the code). The state is made up of a set of constants and variables. Each variable (or constant) is typed and the type may be either an elementary type (Integer, Char, Boolean, LongInt, ShortInt), a structured type (Array, String, Record), or a reference to a constructed type (REF aType).

Two categories of variables are distinguished according to the nature of the objects they designate:

1. either they designate other objects recognised by the object management system (OMS) and are called reference variables (or references) - and the references are typed (to allow static control) by a constructed type,

2. or the objects which they designate are instances of base types (elementary or structured types) and are values which are accessed directly (as in classical programming languages). These objects are not recognised by the OMS.

It will be noticed that the value of a reference variable consists of the value of the OID of the object it designates (this is a generalisation of the idea of a pointer).

Because the distinction between the two categories of variables (reference variables and base type variables) is made by the types which form two disjoint sets (constructed types and base types), the key word REF could have been omitted as in Eiffel. In Guide, we decided to leave it explicit.

Example

```
//declaration of a constructed type variable:
element: REF Element;
//declaration of a simple type variable:
i : Integer;
// declaration of structured type variables:
table : Array[12] OF Boolean;
structure : Record
name : String[20];
element : REF Element;
END;
```

As well as the variables used to define the state of objects, there is another category of variables, called local variables, which are declared in the body of methods and procedures. The methods and procedures parameters are part of the same category.

Since Guide is a persistent object model, the life span of the objects is independent of that of the programs which use them, and consequently of the variables which designate them.

The life span of a variable depends on where it has been declared:

- Declaration in the state (visible variables declared in a type or state variables declared in a class): the variable has the same life span as the object of which it is a part.
- Declaration as a method or procedure parameter or as a local variable: the variable is created at each call and has a life span equal to the execution time of the method or procedure.

4.2 Scope of a variable

The scope of a variable depends on where it is declared.

- Declaration in the type (cf. 5.2): the variable is visible in all the sub-types and all the classes which implement them.
- Declaration in the class (cf. 6.3): the variable is visible in all the sub-classes.

- Declaration as a parameter of a method or a procedure (cf. 6.4): the variable is visible in the body of the method or the procedure. If it is an IN (or IN_OUT) parameter of a method, it is also visible in the activation conditions associated with the method.
- Declaration as a local variable: the variable is visible in the body of the method or procedure.

4.3 Objects and references

A permanent object is an object which must be created and destroyed explicitly. It is only indirectly accessible from a reference variable. The transmission of such an object as a parameter of an operation consists in passing the value of a reference which designates it.

4.3.1 Creation of an object

To create a permanent object, the method New is called for the class of that object. The name of the site where the object is to be created may be given as a parameter to this method. By default, the object is created at the site of execution of the method New. At the call return, a reference to the new instance of the class is returned. The state fields of the object are not initialised except where the initial value has been defined in the type or the class of the object. To create instances of a class with different initial state values, an initialisation method must be defined within the class. Its function will be to initialise the state with the values of the associated parameters. After the creation of an instance (method New in the class, cf. Appendix C.8), the initialisation method with the desired parameters must be called.

Example

```
// declaration of a reference to the object:
a : REF aType;

// calling the method New on the class of the object:
a := aClass.New;

// creation of the object at the site of execution;
a := aClass.New("name_of_site");
// creation of the object at a specified site;

// explicit initialisation of the object
// with the desired values:
a.Init ( ... );
```

4.3.2 Explicit destruction of an object

A permanent object is destroyed by calling the Destroy operation for that object (cf. methods defined for the type Top, cf. Appendix C.9). However, the Guide system uses

a garbage collector [3] to save the programmer from having to remove unused objects explicitly.

Example

```
// let x be the reference to the object to be destroyed;
// calling the method Destroy on the object:
x.Destroy ;
```

4.3.3 Designation of the current object : SELF

The predefined reference variable SELF (or self) always contains the value of the OID of the currently executing object. This variable is used when a method is to be applied to the current object.

4.3.4 Constant reference NIL

The value NIL (or nil) is allocated to a reference variable to express the idea that it does not designate any object. This is what the system does at the Destroy operation - the value NIL is allocated to the reference variable of the destroyed object.

On the other hand, a reference variable not yet initialised has an undefined value. It is not initialised to NIL.

To meet the needs of unrestricted genericity (cf. section 8.1.1), the use of NIL has been extended to simple types. A value corresponding to NIL for each simple type has been chosen, namely: `0` for numerical types, `'\0'` for the type Char and `FALSE` for the Boolean type.

The constant NIL is not typed. Since this value may be assigned to any reference variable, whatever its type, the type which could be attributed to NIL would be a type conforming to all types. It is not possible to define such a type in our model.

4.4 Environment Variables

Some reference variables, called environment variables, can be accessed by the programmer to allow interaction with the execution environment. These variables are maintained by the system or by the shell. When an activity is created, the variables input, output and catal are initialised to the values of the variables corresponding to the creating activity. These can be modified by the programmer, whereas myActivity and myDomain cannot. The operations which are applicable to the objects designated by these references are defined by their type (cf Appendix C). The following predefined variables exist at present:

- input : REF ChanIn, for input operations (cf. Appendix C.6),
- output : REF ChanOut, for output operations (cf. Appendix C.7),
- catal : REF Catal, for access to the name server (cf. Appendix C.4),
- myActivity : REF Activity ; contains the reference to the activity in which the present method is being carried out,

- myDomain : REF Domain ; contains the reference to the domain in which the present method is being carried out (cf. Appendix C.3).

5 Constructed Types

The declaration of a constructed type is made up of a header, a list of synonym declarations (names of types or of classes), a list of visible state variables (cf. 5.2) and a list of method signatures.

Example

```
TYPE Box IS
  // synonyms
  SYNONYM Name = String [80];
  // visible state variable
  owner: Name; // visible state variable
  // methods
  METHOD Put (IN REF Element); SIGNALS full_box;
    // insert an element in the box
  METHOD Take (OUT REF Element); SIGNALS empty_box;
    // extract an element from the box

END Box.
```

Syntax

<constructed_type> = <type_header>
    { <synonyms> }
    { <visible_state_variable> }
    { <method_signature> }
    <end_type>
<type_header> = TYPE <constructed_type_ident>
    [SUBTYPE OF <constructed_type_ident>] IS
<end_type> = END [<constructed_type_ident>] '.'

5.1 Synonyms

Synonym declarations associate one character string with another and facilitate the reading and writing of Guide code. The use of synonyms is restricted to the renaming of types and classes in a type or class declaration.

The effect of a synonym declaration is restricted to the type and its subtypes, to the classes which implement it and to their sub-classes.

Example

```
SYNONYM BitMap = Array [1000,1000] OF Boolean;
SYNONYM MailBoxList = List OF REF MailBox;
```

Syntax

<synonyms> = { SYNONYM <ident> '='
(REF <constructed_type_ident>) |<elementary_type> ';' }

5.2 Visible state variables

A visible state variable of an object is a state variable for which the programmer requests the implicit declaration of the read and write operations having the same name as the variable. As for every method, these operations may be synchronised by associating them with an activation condition (cf. 6.7).

Example

```
TYPE Circle IS
  radius : Integer;
  METHOD Surface : Integer;
END Circle.
```

Use of an object of the type circle:

```
// declarations
i : Integer;
cir : Circle;
...
// instructions
cir.radius := 3;
i := cir.radius;
```

5.2.1 Declaration and initialisation

The declaration of a visible state variable takes the same form as the declaration of a non-visible state variable in the body of the object class (cf. 6.3). If a visible state variable is initialised, each class implementing this type will have the same variable initialised. In the case of a reference variable, the initialisation (i.e. the search for the object with symbolic name given to the right of the '=', and the allocation of its reference to the variable) is made at the creation of each instance. The search for the object is made in the current catalogue, designated by the reference catal. If no object is found, the reference variable is set to NIL. If the object found does not conform to the type of the reference variable, the exception CONFORMANCE_ERROR is flagged.

Example

```
size : Integer = 10;
toto : REF Circle = "my_circle";
                // my_circle is an object of the type Circle
```

5.2.2 Constant visible state variable

If a visible state variable is declared to be a constant, only the read operation is defined; the declaration of such a variable is preceded by the key word CONST and must specify its value. This visible state variable is not modifiable. In the case of a reference variable, this variable always designates the same object.

Example

```
CONST toto : REF Circle = "my_circle";
                // my_circle is an object of type Circle
```

5.2.3 Syntax

<visible state variable> =
    <ident> { ',' <ident> } ':' (<base_type>
        | REF <constructed_type_ident>) ['=' <constant_expr>]
    | CONST <ident> { ',' <ident> } ':' (<base_type>
        | REF <constructed_type_ident>) '=' <constant_expr>

5.2.4 Restrictions

The use of the visible state variables of the type String, Array and Record is restricted as follows :

- The values of variables of type Record can only be read and written.

  Direct access to the sub-fields of these variables is not possible.

- The values of variables of type String and the values of their elements (i.e. characters) can be directly read and written.

- The values of variables of type Array and the values of their elements can be read and written directly. If the elements of the arrays are of the type String, Array or Record, it is not possible to write directly into the sub- elements of these elements.

- It is not possible to initialise (at declaration) variables of types Array or Record, nor to declare them as constants.

5.3 Methods

The set of method signatures defined on an object constitutes the access interface to that object. (Visible state variables are a special case.)

A method signature specifies its name, the list of its argument parameters - preceded by the key word IN, the list of its result parameters - preceded by the key word OUT, the argument/result parameters - preceded by the key word IN_OUT, and the list of exception signals which it can generate - preceded by the key word SIGNALS.

Each parameter is either a typed reference or a simple type variable (base or structured type); In the first case, the type of the reference is specified, preceded by the key

word REF. In the second case only the simple type of the object must be specified. One of the results may be declared as the return value of the method.

Example: two ways of defining the method Get in the type ObjectBox.

```
TYPE ObjectBox IS
  METHOD Put  (IN r : REF Element);
  METHOD Get(OUT r : REF Element); SIGNALS empty_box;
END ObjectBox.

TYPE ObjectBox IS
  METHOD Put (IN r : REF Element);
  METHOD Get : REF aType; SIGNALS empty_box;
END ObjectBox.
```

N.B. : The argument and result identifiers are not analysed by the compiler. Therefore it is possible to write the method signatures as follows:

```
METHOD put (IN REF Element);
```

Syntax

<method_signature> = METHOD <ident>
    ['(' <list_param> ')']
    [':' <return_type>] ';'
    [<list_signals>]
<list_param> = (IN | OUT | IN_OUT) <decl_param>
    { ';' (IN | OUT | IN_OUT) <decl_param> }
<decl_param> =
    [<ident> { ',' <ident> } ':'] <base_type> | REF <constructed_type_ident>
<return_type> = <base_type> | REF <constructed_type_ident>
<list_signals> = SIGNALS <ident> { ',' <ident> } ';'

Methods with the same name

Two methods with the same identifier may be defined in the same type if and only if their respective signatures have a different number of arguments and results or if one has a return value and the other hasn't.

The advantage of this facility is that different method names are not necessary when the function to be defined is the same but the signature differs. For example, the method Get in the preceding example may also be defined with two parameters.

Example

```
TYPE ObjectBox IS
  METHOD Get : REF Element;
    // no arguments, no result, one return value
  METHOD Get (OUT nbRemainingObjects : Integer) : REF Element;
```

```
    // no arguments, one result, one return value
END ObjectBox.

TYPE List IS
  METHOD Next;
    // no argument, no result, no return value
  METHOD Next : REF Top;
    // no argument, no result, one return value
END List.
```

5.4 Construction of a sub-type

The sub-typing relationship may be used to express the specialisation of behaviour. Sub-typing is single. A type can only have one super-type. A sub-type inherits all the definitions of its super-type. It may over-ride these (see below) or supplement them with new synonym definitions, visible state variables or methods.

One type is declared to be the sub-type of another by giving the name of the supertype after the key word SUBTYPE OF on the header line of the specification. If there is no SUBTYPE OF clause in the header, this means that the type is a sub-type of the type Top (cf. Appendix C.9).

Syntax

<type_header> = TYPE <constructed_type_ident>
[SUBTYPE OF <constructed_type_ident>] IS

5.4.1 Overriding of visible state variables

Visible state variables declared in the super-type cannot be overridden.

5.4.2 Signature overriding

Method signatures defined for the super-type may be overridden as long as they continue to conform to the original definitions (see definition of the rule of conformance in section C.9). In particular, it is not possible to add signals. They can only be suppressed.

5.4.3 Example

Example 1

```
TYPE Person IS
  age : Integer;
  name : String[20];
END Person.

TYPE Student SUBTYPE OF Person IS
  university : String[20];
END Student.
```

In this example, sub-typing only relates to visible state variables. The type Student is defined by three attributes: age and name, inherited from type Person, and university which is a new field for this type

Example 2

```
TYPE Account IS
  balance : Integer ;
      // balance of bank account;
  METHOD Credit (IN val : Integer);
      // increase in balance to the value val;
  METHOD Debit (IN val : Integer); SIGNALS operation_impossible;
    // debit operation to the value val; if the account does
    // not have sufficient funds, the exception
    // operation_impossible is generated;
END Account.

TYPE Savings_book_A SUBTYPE OF Account IS
  METHOD Calculate_interest (IN rate of Interest : Integer);
      // calculate interest and update account;
END Savings_book_A.
```

In the example above the type Savings_book_A is a sub-type of the type Account. The type Savings_book_A inherits the visible state variable account as well as the methods credit and debit of the type Account. It also has the method calculate_interest.

6 Constructed Classes

A class is composed of a header, the declaration of state variables (cf. 6.3), the methods code (cf. 6.4) and a control clause (cf. 6.7). In the header of the declaration of a class, the key word IMPLEMENTS allows the specification of the name of the type it implements. If characteristics common to all instances of the class are required, the clause ATTRIBUTE (cf. 6.1) may be used.

In the example below, the two classes LittleBox and BigBox are two implementations of the same type ObjectBox.

Example

```
TYPE ObjectBox IS
        METHOD concat (IN REF ObjectBox) : REF ObjectBox ;
        METHOD put (IN Element) ; SIGNAL box_full ;
        METHOD take (OUT Element) ; SIGNAL box_empty ;
END ObjectBox.

CLASS LittleBox
  IMPLEMENTS ObjectBox IS
  // the box cannot contain more than 5 elements
  // Characteristics of the objects of the class:
```

```
  ATTRIBUTE UNMOVABLE;
  // unmovable objects (they remain at the site where they
  // were created);
  // state variables
  CONST size : Integer = 5 ;
  nb_elt, last, first : Integer ;
  contents : Array [size] OF Element ;
  // methods
  METHOD concat (IN box: REF ObjectBox) : REF ObjectBox ;
                  SIGNAL box_too_small ;
  BEGIN
  ...
  END concat ;

  METHOD put (IN elt :Element) ; SIGNAL box_full ;
  BEGIN
  ...
  END put ;

  METHOD take (OUT elt : Element) ; SIGNAL box_empty ;
  BEGIN
  ...
  END take ;

  // control clause
  CONTROL
    EXCLUSIVE (concat, put, take);
END LittleBox.

CLASS BigBox
  IMPLEMENTS ObjectBox IS
  // the box cannot contain more than 100 elements
  // state variables
  CONST size : Integer = 100 ;
  //a similar sequence to the class LittleBox
END BigBox.
```

Syntax

<class> = <class_header>
[<characteristics>]
{ <synonyms> }
{ <state_variable> }
{ <method> | <procedure> }
[<exception_handling_block>]
[<restoration_block>]
[<control>]

<end_class>
<class_header> = CLASS <class_ident> [SUBCLASS OF <class_ident>]
IMPLEMENTS <constructed_type_ident> IS
<end_class> = END [<class_ident>] '.'

6.1 Characteristics of the objects of a class

In the declaration of a class, the common characteristics of all its objects may be indicated by using the clause ATTRIBUTE. There is at present only one defined attribute : UNMOVABLE. An UNMOVABLE object is an object linked to the site where it has been created and it may not be moved to another site.

Syntax

<characteristics> = ATTRIBUTE <characteristic> { ',' <characteristic> } ';'
<characteristic> = UNMOVABLE

6.2 Synonyms

As in a type, the declaration of a synonym consists of associating one string of characters with another to facilitate the writing and reading of a Guide text.

The effect of synonym declarations in a class is restricted to the class and to all the sub-classes of that class.

6.3 State Variables

The declaration of state variables defines the data template for the instances of that class. The state of an object is composed firstly of visible state variables declared in the type (cf.5.2) and secondly of (non visible) state variables declared in the class. Visible state variables cannot be redeclared.

A state variable allows objects to be designated. The kinds of designation have been defined in section 4, and are as follows: a reference variable of a constructed type to designate objects external to the one in which it is declared; or a simple variable of a base type to designate an internal object (a variable in the sense normally used in programming languages) created statically in the state of the object in which it is declared.

Example

```
owner : REF Person; // reference to an external object;
nb_elt, last : Integer; // variable of the base type Integer;
contents : Array [5] OF REF Element;
                    // variable of base type Array;
```

6.3.1 Initialised State Variables

At declaration, an initial value may be given to the state variables. This value is necessarily a constant and depends on the type of the variable :

- Boolean : `TRUE or FALSE`,
- Character : `'a' or '\078'`,
- Character string : `"string of characters\n"`,
- Whole : `10, 0xa, 012`,
- Reference : `NIL` (cf. 4.3.4), `SELF` (cf. 4.3.3) or "object name". In this case, when the object is created the compiler calls on the name server to initialise the reference.

Example

```
first : Integer = 1;
        // whole variable initialised to the value 1
person1 : REF Person = NIL; // reference initialised to NIL
person2 : REF Person = "toto";
        // reference initialised to the value
        // of the reference of an external object
```

N.B. : at their declaration, arrays or records cannot be initialised.

6.3.2 Constant State Variables

The declaration of a constant is similar to that of an initialised variable preceded by the key word CONST. The rules of declaration and use are the same.

The effect of a reference variable being declared constant is that it will never designate any other object than the one it designates at initialisation.

Example

```
CONST size : Integer = 5;
  // constant of the base type Integer;
CONST largesize : Integer = size*10;
  // correct declaration if 'size' has been defined;
CONST person : REF Person = "toto";
  // constant reference initialised to an external object;
CONST size : Integer = 5 + 0x10 + 077;
  // constant variable of the class Integer;
```

N.B. : the declaration of constant arrays or records is not advantageous since they cannot be initialised at declaration.

6.3.3 Syntax

<state_variable> =
    (<ident> { ',' <ident> } ':' <base_type>
        | REF <constructed_type_ident> ['=' <>constant_expr])
    | (CONST <ident> { ',' <ident> } ':' <base_type>
        | REF <constructed_type_ident> '=' <constant_expr>)

6.4 Methods

Methods define the access points for the objects of the class in which they are declared. The declaration of a method consists of: a header; the list of declarations of its local variables; and the list of instructions which make up its body. The declaration header consists of the method signature in which the names of the arguments, the results and the parameters which are at the same time arguments and results must appear. In addition, a block of exception handling may be defined at the end of the body of the method to handle all the exceptions created by the set of instructions of the body.

Example

```
METHOD Put (IN elt : REF Element) ; SIGNALS full_box ;
BEGIN
  IF nb_elt < size THEN
    contents[last] := elt ;
    last := (last+1) MOD size ;
    nb_elt := nb_elt + 1 ;
  ELSE RAISE full_box;
  END ;
END Put ;
```

Syntax

<method> = <method_header>
    { <local_variable> ';' }
    BEGIN
    <instructions_list>
    [<exception_handling_block>]
    [<restoration_block>]
    END [<ident>] ';'
<method_header> = METHOD <ident>
    ['(' <param_list> ')']
    [':' <return_type>] ';'
    [<signals_list>]
<local_variable> = <state_variable>

6.5 Procedures

Procedures allow the factorisation of the code needed to write the body of methods defined for a class. A procedure is made up of: a header which describes its name and its signature; a list of declarations of local variables; and a body composed of a list of instructions.

The procedures must be declared at the same level as the methods. As with methods, the argument/result parameters and a block of exception handling associated with the body of the procedure may be declared.

Syntax

```
<procedure> = <procedure_header>
        { <local_variable> ';' }
        BEGIN
        <instructions_list>
        [ <exception_handling_block> ]
        [ <restoration_block> ]
        END [<ident>] ';'
<procedure_header> = PROCEDURE <ident>
        [ '(' <param_list> ')' ]
        [ ':' <return_type> ] ';'
        [ <signals_list> ]
```

6.6 Paramater Passing

In Guide, all parameters are passed by value according to one of the three modes IN, IN_OUT or OUT. The mode 'return parameter' is a specific case of the mode OUT.

Restriction The use of the same variable as an actual parameter of several OUT (or IN_OUT) parameters during one method or procedure call is prohibited. This limitation arises from the fact that it is not possible to ensure that a local execution behaves in the same way as a remote execution.

Syntax

```
<param_list> =
        ( IN | OUT | IN_OUT ) <param> { ';' ( IN | OUT | IN_OUT ) <param> }
<param> =
        <ident> { ',' <ident> } ':' (<base_type> | REF <constructed_type>)
<return_type> = (<base_type> | REF <constructed_type>)
```

6.6.1 IN Parameters

A parameter of type IN consists of a variable local to the procedure or method initialised to the value of the actual parameter. The example below shows the difference between the two:

Example

```
PROCEDURE Proc (IN x : Integer);
BEGIN
 x := x + 1;
END Proc;
...
a := 0;
Proc (a); // a = 0 (and not 1) after the procedure call proc.
```

6.6.2 OUT Parameters

A parameter of type OUT consists of a variable local to the procedure or method which is not initialised. At the end of the procedure or method the value of this variable is copied back to the actual parameter. The example below shows the effect of this choice :

Example

```
PROCEDURE Proc (OUT x : Integer);
BEGIN
 x := 10;
END Proc ;
...
a := 0;
Proc (a) ; // a = 10 after the procedure call proc.
```

6.6.3 IN_OUT Parameters

A parameter of type IN_OUT consists of a variable local to the procedure or method initialised to the value of the actual parameter. At the end of the procedure or method the value of this variable is copied back to the actual parameter. The example below shows the effect of this choice :

Example

```
PROCEDURE Proc (IN_OUT x, y : Integer);
BEGIN
 x := x + 1;
 y := x;
END Proc ;
...
a := 0; b := 3;
Proc (a, b); // b = 1 (pas 3) after the procedure call proc.
```

6.6.4 Return Parameter

The return parameter allows a method or procedure to be used as a function. See instruction RETURN (cf. 7.13) for its use.

Example

```
PROCEDURE Double (IN x : Integer) : Integer;
BEGIN
 RETURN 2*x;
END Double;
```

6.7 Activation Conditions

The control clause determines the conditions of synchronisation between the method calls for each object. For example, one could specify that methods must be executed in mutual exclusion. The conditions of synchronisation are Boolean expressions, which are functions of the state of the object, the input parameters of the method and the synchronisation counters associated with each method. These counters are also visible in the methods and procedures code of the class. The utilisation syntax is:

counter_name (method_name).

The defined counters are as follows :

- invoked(m) : total number of calls on the method from the creation of the object,
- started(m) : number of calls on the method which have been accepted (not blocked),
- completed(m) : number of completed executions,
- pending(m) : number of activities awaiting execution of the method,
- current(m) : number of calls in course of execution.

The first three counters (invoked, started, completed) are created and initialised to 0 at the creation of an object. Their value can only increase during the life of the object. The two other counters (pending and current) are calculated from the first with the relationships:

pending(m) = invoked(m) - started(m),

current(m) = started(m) - completed (m).

A control clause is associated with each method and all the clauses are grouped together at the end of the class after the key word CONTROL. As this association is made by giving the name of the method, it is not possible to define different control clauses for methods of the same name; the same clause applies to all the methods with the same name. In addition, the only input parameters which may appear in the common activation condition are those which are common to all the methods.

A control clause may be associated in the same way with the visible state variables since each one corresponds to one read and one write method. As these two methods are only defined syntactically by the name of the visible variable, only a single control clause can be defined for both operations.

The programmer can use certain key words for the most frequently encountered conditions :

- EXCLUSIVE (m1, m2, ..., mk) : to designate the condition of mutual exclusion between the methods mi, i.e. : Σcurrent(mi) = 0.

- READER (m1, m2, ..., mk), WRITER (m'1, m'2, ..., m'l) : to designate that the methods mi and m'j will be used with a policy of reading/writing.

  A method with the characteristic READER only executes when no method with the characteristic WRITER is executing.

  A method with the characteristic WRITER only executes when no other WRITER and READER method is executing.

There may be several EXCLUSIVE clauses in a control block. If there is a READER clause, there is an obligatory WRITER clause (and vice versa). There may be more than one group of READER/WRITER clauses.

Overriding methods automatically inherit the activation condition defined for the method of the same name in the super-class. However, it is possible to redefine the activation conditions in the sub-class (cf. 6.8.4).

When no clause is defined for a method, there is no synchronisation constraint on calling that method.

Syntax

<control> =
    CONTROL { <predefined_condition> | <non_predefined_condition> ';' }*
<non_predefined_condition> =
    <ident> { ',' <ident> }* ':' < boolean_expression>
<predefined_condition> =
    EXCLUSIVE '(' <method_list> ')'
    | WRITER'(' <method_list> '),' READER '('<method_list> ')'
    | READER '(' <method_list> '),' WRITER'('<method_list> ')'

Example

```
CLASS FixedSizeBuffer
 IMPLEMENTS ProducerConsumer IS
 CONST size : Integer = 10;
 buffer : Array [size] OF REF Element;
 first, last: Integer = 0;

 METHOD Put(IN m: REF Element);
  BEGIN
  buffer[last]:=m;
  last:=(last+1) MOD size;
  END Put;

 METHOD Get(OUT m: REF Element);
  BEGIN
  m:=buffer[first];
  first:= (first +1) MOD size;
  END Get;
```

```
 CONTROL
  Put: (completed(Put) - completed (Get) < size) AND current(Put)=0;
  Get: (completed (Put) > completed (Get)) AND current(Get)=0;
END FixedSizeBuffer.
```

6.8 Construction of a sub-class

The construction of a class by inheritance from another enables the specialisation of behaviour expressed by their types by the specialisation of the state model and the set of methods to be completed. It is also a way of sharing a common implementation (between a class and its sub-classes). The type implemented by a sub-class of a given class is therefore necessarily the sub-type of the type implemented by that class. Inheritance is single - a class has at the most one super-class.

Syntax

<class_header> = CLASS <class_ident>[SUBCLASS OF <class_ident>]

When the clause SUBCLASS OF does not exist, the class is a sub-class of the class Top which implements the type Top (cf. Appendix C.9).

Inheritance affects all the defined elements of an inherited class (parent class or super-class): state variables (visible or not), methods and control clause, and blocks linked to exceptions : the handling block EXCEPT and the restoration block RESTORE. Conversely, procedures are not inherited.

The over-riding of an inherited method may be achieved in different ways: code over-riding without signature over-riding, signature and code over- riding, over-riding of the control clause associated with the method, and over-riding of clauses linked to exceptions.

6.8.1 Over-riding of the state model

The definition of new state variables is possible. State variables (visible or not) defined at the level of the parent class are inherited, but over-riding is not allowed.

Example

```
CLASS Figure
 IMPLEMENTS Figure IS
  // state variables
  view : REF BitMap ;
END Figure.

CLASS Circle SUBCLASS OF Figure
 IMPLEMENTS Circle IS
  // state variables - equivalent to the inheritance model
  centre : REF Point ;
  radius : Integer ;
END Circle.
```

6.8.2 Over-riding of a signature

Signature over-riding appears in the type and causes a redefinition of the body of the method within the class.

6.8.3 Code over-riding

Over-riding of the body of a method is specified independently of the overriding of its signature.

In the new body of a method, it is never possible to call a procedure defined in the parent class. Procedures allow a code factorisation which is not visible outside the class in which they are defined.

In the definition of a sub-class of a given class, it is possible to call a method of the super-class which is over-ridden in the current class by the use of the key word SUPER. This allows code sharing betweeen parent and child classes even in the case of over-riding.

Example

```
CLASS Refbib
IMPLEMENTS Refbib IS
  numref : Integer;
  name : String[15];

  METHOD Print ;
  BEGIN
    output.WriteInteger (numref);
    output.WriteString (name);
  END Print ;
END Refbib.

CLASS Refbook SUBCLASS OF Refbib
IMPLEMENTS Refbook IS
  editor : String[10]; // override the state;

  METHOD Print ;     // override the print method;
  BEGIN
   SUPER.Print;
   // printing of the fields defined in the class Refbib
   // by calling the print method of the super-class (Refbib);
   output.WriteString (editor);
  END Print ;
END Refbook.
```

6.8.4 Over-riding of the control clause

The control clause defined in the super-class is automatically inherited by the sub-class. However, it is possible to have a new control clause in the sub-class, firstly to define

the activation condition of the new methods of the sub-class, and secondly to change the activation condition of the inherited or over-ridden methods.

When an activation condition of an inherited or over-ridden method is redefined in the sub-class, it completely replaces that of the super-class. There is no addition of the condition of the sub-class to that of the super-class.

6.8.5 Over-riding of the blocks EXCEPT and RESTORE

A sub-class always inherits the EXCEPT blocks defined in its super-class. It may however define some blocks locally : the handlers of the new EXCEPT clause will have priority over those defined in the super-class.

For the RESTORE blocks, the same over-riding principle as for the control clause applies : they are inherited except if they are redefined in the sub- class (a new RESTORE block completely replaces that of the super-class).

7 Instructions

An instruction may be an assignment, a method or procedure call, a conditional or iterative instruction, the parallelism instruction, instructions for dynamic conformance checking, or the instruction return (for exceptions : RAISE, or normally : RETURN). Note that semi-colons are used as instruction terminators rather than as separators. In fact, each instruction must be followed by a semi-colon.

Syntax

```
<list_of_instructions> = <instruction> { <instruction> }
<instruction> =
      ( <assignment>
      | <method call>
      | <procedure call>
      | <IF_instruction>
      | <CASE_instruction>
      | <TYPECASE_instruction>
      | <ASSERTYPE_instruction>
      | <FOR_instruction>
      | <WHILE_instruction>
      | <REPEAT_instruction>
      | <CO_BEGIN/CO_END_instruction>
      | <BLOCK_instruction>
      | <RETURN_instruction>
      | <RAISE_instruction>
      | BEGIN <list_of_instructions> [ <exception_handling_block> ] END
      ) ';'
```

7.1 Assignment and expressions

Assignment is the operation which modifies the value of a variable. An assignment instruction takes the following form : <variable> ':=' <expression>.

The semantic validity of an assignment instruction depends on the nature of the variable identified by the designator, reference, internal object, local object or pointer and on the nature of the value produced by the evaluation of the expression given on the right. This validity is checked statically by

the compiler, using type identity for simple types and conformance(see 9) for reference variables. Although this static verification mechanism is useful for detecting errors as soon as possible, it is not sufficient in certain cases and a type check may be carried out dynamically with the ASSERTYPE instruction (cf. 7.7).

For base types, Guide offers an extension to the identity rule allowing the assignment of an integer to a character (and vice versa) , of an integer to a ShortInt (and vice versa), of an array of elements of the same type but different size, etc. The assignment rules for base types are defined in Appendix B sections 1.6 and 2.4.

Note: According to the above rules, it is not possible to assign a reference variable to a simple type variable.

Example

```
toto := titi ;
// valid if toto and titi are either references such that the
// type titi is in conformance with that of toto, or they are
// identical (or compatible) simple type variables.
bool := bill or ball ;
// bool, bill and ball are Boolean variables.
toto := titi.AMethod (a,b) ;
// the method 'Amethod' has a return parameter of a type
//  compatible with the type of the variable toto.
```

Syntax

<assignment> = <variable> ':=' <expression>
<expression> = <ident> | <method call> | <procedure call>

7.2 Method Call

The method call for an object is the key instruction of an object-oriented language such as Guide. Such an instruction takes the following form : toto.op (actual parameters). This means the application of the method op to the object designated at the time of execution by the reference variable toto.

If the method possesses a return value the call instruction is an expression.

The validity of a method call instruction depends on the type of the reference variable toto (the method called must be specified in the type) and on the validity of the parameter passing (the assignment rules defined above in 7.1 apply).

Syntax

<method call> =
    <reference_variable>'.'<ident> ['('<expression> { ',' <expression> } ')']

7.3 Procedure Call

The procedure call is an instruction which allows the factorisation of code in the body of a class. A procedure call instruction takes the following form : <procedure_name> ['(' <parameters> ')'].

The validity of a procedure call is checked by the compiler which verifies whether the procedure (the execution of which is) requested has been declared and if the parameters have been passed correctly. (The assignment rules defined above in 7.1 apply.)

Syntax

<procedure call> = <ident> ['('<expression> { ',' <expression } ')']

7.4 Instruction IF

Example

```
IF bool THEN <an action> ;
ELSE <another action> ;
END ;
```

Syntax

<IF_instruction> =
    IF <Boolean_expression> THEN
    <list_of_instructions>
    [ELSE <list_of_instructions>]
    [<exception_handling_block>]
    END

7.5 Instruction CASE

The CASE instruction specifies the selection and execution of a list of instructions as a function of the value of an expression

Syntax

<CASE_instruction> =
    CASE <expression> OF
    { <tag_list> ':' <list_of_instructions> END ';' }*
    [OTHERS ':' <list_of_instructions>]
    [<exception_handling_block>]

END

The expression in the CASE instruction must be of type Boolean, Char, Integer, ShortInt, LongInt or String. The tags are constant expressions of the same type as the expression of the instruction. Only the list of instructions corresponding to the tag of the same value as the expression is executed. If it is supplied, the list of instructions corresponding to the tag OTHERS is executed when no other tag has the same value as the expression.

Example

```
CASE i OF
  1:  <an action> ;
    END ;
  2,3: <another action> ;
    END ;
  OTHERS : <a third action> ;
END ;
```

7.6 Instruction TYPECASE

The TYPECASE instruction specifies the selection and execution of a list of instructions as a function of the type of the designated object given as selector by the reference. This instruction may not be applied for base type variables.

Syntax

<TYPECASE_instruction> =
TYPECASE <reference_variable> OF
{ <type_ident> ':' <list_of_instructions> END ';' }*
[OTHERS ':' <list_of_instructions>]
[<exception_handling_block>]
END

The principle of selection and execution is the same as for the CASE instruction. Instead of relating to the value of the variable, the selection relates to the type of the reference variable: the instruction sequence is carried out if the type of the object designated by <reference_variable> is equal to that of <type_ident>, the clauses being evaluated sequentially. Within each group of instructions selected by the type <type_ident>, the reference <reference_variable> which acts as selector is considered to be of the type <type_ident>. For the OTHERS clause, this reference is used with the type defined at the declaration and not with the type of the object actually designated.

Example

```
TYPECASE ref_figure OF
  Figure : ref_figure.Display (abscissa, ordinate);
```

```
        // call 'Display' on the type 'Figure'
        END ;
    Circle : ref_figure.Display (centre, radius);
        // call 'Display' on the type 'Circle'
        END ;
    OTHERS : <error handler> ;
END ;
```

7.7 Instruction ASSERTYPE

This instruction takes the form : x := ASSERTYPE (y). It allows a request for the execution of a dynamic type check between the type of the object designated by the reference y and the type of the reference variable x. In addition, if the types are compatible, the assignment x := y is made. The usefulness of such an instruction is shown in the example below.

Example

```
ref1 : REF Document;
...
ref1 := catal.Search ("Note54");
```

As the object returned by the method "Search" is declared in the type "Top" (see the methods defined on the class Catal Appendix C.4), the compiler does not permit the above assignment because the object returned could be of any type. However, it would be preferable if the assignment was permitted in the case where the object is of type Document (or a type conforming to Document). To resolve this problem, the instruction ASSERTYPE is used :

```
ref1 : REF Document ;
...
ref1 := ASSERTYPE (catal.Search("Note54"));
```

The ASSERTYPE instruction has an input parameter which is an expression of the type reference and a return parameter which is of the type reference. In addition it may flag the exception 'CONFORMANCE_ERROR'. On execution, the code generated by the compiler will evaluate the input expression and verify that its type is the same as (or conforms to) that of the reference on the left side of the assignment. If yes, the assignment of references is carried out; if no, the exception 'CONFORMANCE_ERROR' is flagged and the value NIL is assigned to the reference.

Limitation In version V 1.5, the conformance checking carried out by the compiler at the ASSERTYPE instruction is limited to the sub-typing relationship.

Syntax

<ASSERTYPE_instruction> =
    <reference_variable> ':=' "ASSERTYPE" '(' <expression> ')'

7.8 Instruction FOR

Example

```
FOR i := 1 TO 10 BY 2 DO
  <an action> ;
END ;
```

Syntax

<FOR_instruction> =
    FOR <ident> ':=' <integer_expression> TO <integer_expression>
        [BY <step>] DO
        <list_of_instructions>
        [<exception_handling_block>]
    END

7.9 Instruction WHILE

Example

```
WHILE  bool DO
  <an action> ;
END ;
```

Syntax

<WHILE_instruction> =
    WHILE <Boolean_expression> DO
    <list_of_instructions>
    [<exception_handling_block>]
    END

7.10 Instruction REPEAT

Example

```
REPEAT
  <an action> ;
UNTIL (i = 0);
```

Syntax

<REPEAT_instruction> =
    REPEAT
    <list_of_instructions>
    [<exception_handling_block>]
    UNTIL <Boolean_expression>

7.11 CO_BEGIN Instruction

The CO_BEGIN/CO_END_instruction initiates several activities in parallel, the parent activity being suspended during the execution of this instruction. This instruction possesses a termination expression which specifies the conditions under which the instruction ends. The body of each of the child activities consists of a single instruction restricted to the method call of an object.

Syntax

<CO_BEGIN/CO_END_instruction> =
CO_BEGIN
{ <ident> ':' <method call> ';' }*
[<exception_handling_block>]
CO_END [<Boolean_expression> | ALL | FIRST]

A Boolean variable is associated with the name of each activity and takes the following values :

- it is indeterminate as long as the activity is executing,
- it becomes false as soon as the activity terminates abnormally (i.e. it raises an exception),
- and it becomes true as soon as the activity terminates correctly.

The parent activity, which was suspended at the creation of the child activities, is reactivated as soon as the Boolean expression becomes true. The child activities which are still active are automatically destroyed. The key words ALL and FIRST allow a more succinct form than the Boolean expression of termination.

If the termination condition is not specified, the parallel instruction terminates when all the child activities terminate correctly (equivalent to the key word ALL).

If a child activity terminates with an exception (i.e. the associated Boolean variable becomes false) making the Boolean expression of CO_END false, the instruction CO_BEGIN/CO_END terminates with an exception (for more details on termination of the instruction CO_BEGIN/CO_END with an exception, see 7.14.5).

Example A simultaneous execution of the methods print and duplicate on the same object designated by a reference with the name report may be carried out by the following block :

```
CO_BEGIN
   printing : report.Print;
   duplication : report.Duplicate;
CO_END printing AND duplication;  // equivalent to CO_END ALL;
```

The parent activity continues its execution if the two activities print and duplication terminate correctly.

The parallel block will terminate abnormally (with an exception) if one or other of the two activities terminates with an exception.

7.12 BLOCK Instruction

This instruction packages a piece of code with its execution context and assigns it to a reference variable of type Block (cf. Appendix C.1), in order to be able to use it later (notably when passing the variable in parameter) - either directly by requesting the evaluation of the block with the instruction <block ref>.Value, or by requesting a domain to execute this block with the instruction <domain ref>.Start(<block ref>).

The execution context of a block must not create any side effects when it is executed. On the other hand, since this block could be executed by any method (and thus any object) which had obtained its reference parametrically, it cannot have access to the state variables of the object in which it has been created otherwise the principle of encapsulation of objects would be violated. Following these remarks, the rules defining the execution context of a block are as follows :

Rules of Use

- The local variables of a block are obtained by duplication from the local variables of the method in which it is declared, and the input parameters of the method (IN or IN_OUT). These variables are initialised to the values they have at the time of creation of the block.
- It is also possible to declare local variables in the block.
- The OUT parameters and the method signals are not part of the context of the block. The compiler verifies that they are not used in the block code.
- The same is true for the state variables of the object (except for visible state variables, which, like all the methods of the object, may be accessed with the reference SELF).
- A block cannot be declared in a procedure.

The definition of a block corresponds to the definition of a class consisting of two methods : one method Init executed at the creation of the block to initialise the context, and one method Value for its execution.

The handling of the exceptions generated inside a block can only be carried out by handlers defined in the block or those defined at the level of the class.

Syntax

```
<BLOCK_instruction> =
      <ident> ':=' BLOCK
            {<local_variable>}
            BEGIN
            <list_of_instructions>
            [<exception_handling_block>]
            END
```

Example

```
TYPE T IS
x : Integer;
METHOD M (IN i : Integer; IN_OUT io : Integer; OUT o : Integer);
METHOD N (...);
END T;

CLASS C IMPLEMENTS T IS
y : Integer;
METHOD M (IN i : Integer; IN_OUT io : Integer; OUT o : Integer);
z : Integer;
r : REF D;
bloc : REF Block;
  // block is a reference variable of the type Block;
BEGIN
... // method code;
z := 2;
bloc := BLOCK  // creation of the block and its context;
  // the context is composed of the variables i, io and z;
   t : Integer;  // variable local to the block;
   BEGIN         // block code;
   t := 0;       // allowed;
   t := z;       // allowed (z equals 2);
   z := 0;       // allowed (the variable z of the method M
                 // is unchanged) ;
   i := i+1;     // allowed;
   o := z;       // prohibited;
   x :=1;        // prohibited;
   y := 1;       // prohibited;
   SELF.x := 1;  // allowed because x is a visible variable.
   SELF.y :=1;   // prohibited;
   SELF.N;       // allowed;
  EXCEPT
  ...            // block of exception handling;
  END;           // end of the instruction BLOCK
...  // list of the code of the method M;
r.op (..., bloc, ...);
  // parameter passing of the block reference;
END M ;
...  // code of the other methods;
END C.

Class D IMPLEMENTS D IS
//Code of the method op;
METHOD op (..., IN b : REF Block, ...);
BEGIN
...
```

```
b.Value;  // using the block whose reference has been passed;
          // the code of the block is executed with the
          // context defined at the creation of the block;
...
END op;
END D.
```

7.13 Instruction RETURN

The RETURN instruction terminates a method or procedure. If the method or procedure has a return value, this is specified by the expression which follows the key word RETURN.

Example

```
METHOD DoNothing;
i : Integer;
BEGIN
  RETURN;
  i := 10; // instruction never reached
END DoNothing ;

METHOD ReturnsTwo : Integer ; // method in the form of a function
BEGIN
  RETURN 2 ;
END ReturnsTwo ;
```

Syntax

<RETURN_instruction> = RETURN [<expression>]

7.14 Instructions relating to exceptions

7.14.1 Description of the exception mechanism

Principle The exception mechanism proposed in Guide [2] allows each method to signal, with the instruction RAISE, an exception which is part of its signature. When an exception is flagged, the method which signals it may eventually execute a block of restore instructions (defined by the key word RESTORE, cf. section 7.14.6) before terminating and returning control to the calling method. For the signalling operation, the instruction RAISE has the same semantics as the RETURN instruction.

```
Example :
METHOD M (...); SIGNALS error;
BEGIN
  ...
  IF cond THEN RAISE error;
  ...;
```

```
RESTORE
  // handling is executed immediately before returning control
  // to the caller at the execution of the RAISE instruction.
END M;
```

To every method return terminating in an exception, the calling operation may associate a special handler (defined by the key word EXCEPT), which is executed before passing to the following instruction.

```
Example :
// call method M defined below
BEGIN
  toto.M (...);
EXCEPT
  error : output.WriteString("it is an error");
END;
```

Handler

A handler is a Guide instruction, specifically a block of instructions BEGIN ... END.

The code of the handler must be considered as equivalent to the code of the instructions it protects. It may thus access all the state variables of the object. When it is declared in a method or a procedure, access to the local variables should also be possible ; however, this is prohibited in the current version (for implementation reasons). If the handler contains the instruction RETURN, this is equivalent to a RETURN executed by the method or the procedure in which the handler is defined. Therefore the parameter (if there is one) returned by RETURN must conform to the formal return parameter defined in the header of the method or the procedure. For this reason, the instruction RETURN cannot appear in a handler defined at the level of the class.

Finally, the handler may use the two privileged instructions REPLACE and RETRY explained in detail in 7.14.2.

Handling Block

The handling block is a set of handlers beginning with the key word EXCEPT, each handler being associated with a list of exception masks. The block is associated with a set of instructions, each one capable of signalling one or more exceptions. The masks associated with a handler of the block specify among all the possible exceptions those which the handler can handle. The

handler protects the code with which it is associated from the exceptions which it can handle.

An exception mask comprises a mandatory part giving an exception name 'exc_name', and two optional parts giving a type name 'type_name' and an operation name 'op_name'. It indicates that the handler can be called to handle an exception of the name 'exc_name', this exception having been signalled by a call on an operation called 'op_name' on an object of the type 'type_name' (or on one of its sub-types). The key word ALL may be used for each of these three parts. ALL is the default value for the two optional parts.

The key word SYSTEM may be used instead of the type name. It allows handling of the exceptions signalled by the system. ALL for the type includes the system exceptions.

Example

```
EXCEPT
  exc1,exc2 : <handler common to the exceptions exc1 and exc2>;
  exc3 FROM Atype : <handler for the exception exc3 signalled
                    by an operation call carried out on an
                    object of the type (or sub-type of) A type>;
  exc4 FROM M1.Atype : <handler for exc4 signalled by a call
                       on the operation M1 carried out on an
                       object of the type (or sub-type of)
                       Atype>;
  ALL : <handler common to all the exceptions>;
 END;
```

Association It is possible to associate an exception handling block with the set of instructions of a block structure, a method or procedure, or a class. The block handlers are then associated with each one of the operation calls of that set. The block associated with an IF, FOR, ..., also manages the test expression.

If the call on an operation signals an exception and several handlers, with adapted masks, protect that call, the first of the handlers defined in the same block, or the handler of the most internal of the blocks in the case of embedded blocks, is chosen.

If no handler is ready to handle a signalled exception, the calling operation is terminated and the exception system signals UNCAUGHT_EXCEPTION. This exception will be recoverable in the caller's caller by a handler associated with the mask UNCAUGHT_EXCEPTION (FROM SYSTEM).

7.14.2 Specific Instructions

After the execution of the handler, the caller retrieves from the next operation the call of the signalling operation. If the signalling operation has to return a value used in an expression, the next operation is the evaluation of the expression. The handler may then provide a replacement value for the expected return value using the instruction REPLACE. This instruction has the same syntax as the instruction RETURN.

The replacement value provided must conform to the type of the return value of each of the operations likely to signal an exception handled by the handler. Verification of this conformance is not made in version 1.5 of the compiler.

The handler also has at its disposal the instruction RETRY which asks the caller to re-execute the signalling operation rather than to restart the next instruction.

Example

```
CLASS C IMPLEMENTS S IS
toto : REF C;
y : Integer;

METHOD M : Integer; SIGNALS error1, error2;
...
END M;
```

```
// method calling the method M
METHOD N;
x : Integer;
BEGIN
  x := toto.M(y);
  EXCEPT
  error1 : BEGIN y := 20; RETRY; END; // reexecution of the call;
  error2 : REPLACE 0; // the return value of M is 0;
END N;
```

7.14.3 Exceptions in a handler

The handler may signal an exception or declare handling blocks in the same way as the code which it protects. The signalling of an exception by the handler is equivalent to the signalling of the exception by the method or procedure in which it is declared, i.e., one exits not only from the handler but from the operation currently executing. The exception must therefore be declared in the signature of the operation.

In the present version, it is not possible to signal an exception in the handling block associated with the class. (It would be necessary to verify that the exception has been declared in the signature of each operation during the execution of which the handler is likely to be called.) This does not occur in version 1.5 of the compiler.

The instructions of the handler benefit from the same protection as the signalling operation, except that it cannot cause itself to be called (to avoid recursion). It may in addition define supplementary handling blocks, which would then have the highest priority. In particular the RETRY instruction may benefit from the protection of the supplementary handlers called first.

7.14.4 Specific Cases of Signalling

When a method (used in an expression) is a function (i.e. it has a return parameter) and signals an exception, the expression continues executing after the execution of the handler associated with the exception. This is also true when an actual parameter of a method call is itself a method call (x.op1(y.op2(...), ...)). This implies that the return value given after an exception is not significant (the RETURN instruction having been unable to execute before RAISE) and thatthe handler associated with this exception must comprise the instruction REPLACE to validate the continuation of the execution of the expression.

7.14.5 Exceptions and CO_BEGIN

If an operation of a branch of the parallel block flags an exception, the activity is stopped. Nevertheless, the other branches are not automatically stopped.

The termination function CO_END of the parallel block expresses the correct termination condition of the block on the correct termination of each of its branches, using the Boolean operators AND and OR. This construction allows two different processes to search in parallel for the same result and to stop when one of them has found it.

A branch terminates correctly if and only if it does not flag (propagate) an exception. If the correct termination of a branch satisfies the block termination function, the branches still active are stopped and the CO_BEGIN is exited normally. If incorrect termination (with an exception) of a branch prohibits the correct termination of the parallel block (the Boolean expression remains false), the branches still active are stopped and the CO_BEGIN is exited, flagging the system exception JOIN_FAILED. This exception may be recovered in the caller by a handler associated with the mask JOIN_FAILED FROM SYSTEM.

Note : the body of each branch of a CO_BEGIN (which consists of a method call) is only protected by the handlers defined in a handling block of the class.

7.14.6 Restoration

A restoration block may be associated with a method or procedure, or with a class. It is a list of Guide instructions of any kind, beginning with the key word RESTORE. The block associated with the method may use the local variables of the method. The block associated with the class may only use the state variables of the class.

When a method signals or propagates an exception, it executes firstly the eventual restoration block which is associated with it, then the eventual block associated with the class. Only then is the control given to the calling method which executes the handler associated with the exception.

In the case of a procedure, only the restoration block associated with the procedure is executed. The entire block associated with the class is not.

The restoration block associated with a method is not inherited, but one associated with a class is implicitly inherited if the sub-class has no restoration block associated with it.

The exceptions are manipulated in a restoration block in the same way as they are outside it. The class handlers protect the instructions of the restoration blocks associated with the method or the class. At the same time, if an exception is signalled or propagated by the restoration block itself, the exception system RECOVERY_FAILED is propagated. It is assumed that the object could not be put back into a coherent state.

7.14.7 Interruptions

There are two ways of stopping a Guide program from the outside. They correspond to the two Unix signals SIGINT and SIGQUIT. The first is generated by a ^C, which produces an abrupt interruption of the process. The second is generated by a ^|, but may be redefined for example by the Unix instruction "stty quit ^A". This causes a gradual interruption of the activity, invoked by the exception system QUIT (FROM SYSTEM).

The exception system QUIT is handled like all the other exceptions. If it is not handled it propagates itself into the exception system UNCAUGHT_EXCEPTION, which will normally terminate the activity after having unstacked each embedded call, and thus after having executed all the defined restoration blocks.

For implementation purposes, the exception QUIT is not immediately generated. This happens in particular if the program loops during calculation instructions without making a call to the system (no GuideCall).

Example

```
CLASS C IMPLEMENTS C IS
finished : Boolean;
METHOD M;
BEGIN
  finished := FALSE;
  REPEAT
  output.WriteString("type ^| to stop\n");
  EXCEPT
    QUIT FROM SYSTEM : finished := TRUE;
  UNTIL finished;
END M;
```

7.14.8 System Exceptions

Apart from the exceptions UNCAUGHT_EXCEPTION, CONFORMANCE_ERROR, RECOVERY_FAILED, JOIN_FAILED and QUIT already mentioned, there are other exceptions which may be flagged by the system. All the exceptions flagged by the system may be collectively designated by the mask ALL FROM SYSTEM.

7.14.9 Syntax

Signalling

<RAISE_instruction> = RAISE <exception_ident> ';'

Handling

<exception_handling_block> =
    EXCEPT { <exception_mask> : <handling> }*
<exception_mask> = (<exception_ident> | ALL)
    [FROM (<constructed_type_ident>["."<method_ident>]
    | SYSTEM | ALL)]
<handling> = <instruction> ';'

Restoration

[<restoration_block>] = RESTORE <list_of_instructions>

7.14.10 Example

```
CLASS Buffer IMPLEMENTS Buffer IS
CONST max : Integer = 100;
buffer : Array[max] OF REF Element;
n : Integer = 0;
```

```
METHOD Put (IN elt : REF Element); SIGNALS Full;
BEGIN
  IF (n < max)
  THEN
    buffer[n] := elt;
    n := n+1;
  ELSE
    RAISE Full;  // signalling the exception
  END;
END Put;

METHOD Get : REF Element; SIGNALS Empty;
BEGIN
  IF (n > 0)
  THEN
    n := n-1;
    RETURN buffer[n];
  ELSE
    RAISE Empty;  // signalling the exception
  END;
END Get;
END Buffer.

// example of calling the method Get:
METHOD Clear_buffer (IN b : REF Buffer);
x : REF Element;
BEGIN
   REPEAT
     x := b.Get;
     x.Destroy;
   UNTIL false;
   EXCEPT         // handling block associated with the method;
     Empty : RETURN;
END Clear_buffer;
```

When the signal Empty is returned by the method call b.Get, the handler associated in the EXCEPT clause is executed, i.e. in this case the RETURN instruction. The effect of this is to terminate the method Clear_buffer.

8 Genericity

The introduction of genericity into a language offers the programmer a way of factorising sub-programs by parametrising them with the types of the data to which they are applied. In Guide, genericity allows the definition of parametrised type and class constructors which will be used to define Guide types and classes.

Guide offers a non-restricted genericity mechanism: the formal parameter of genericity is not typed; and a restricted genericity mechanism: the formal parameter is typed

in such a way as to impose and check statically that the effective parameter conforms to the type of the formal parameter. The advantage of restricted genericity is that it allows the use of methods on generic type variables.

Definition : a generic type variable is a variable declared in a generic type or class the type of which is one of the parameters of genericity.

In class constructors, a parameter of constant size of type <Integer> is initialised at the creation of an object of this class.

The declaration of a generic type resembles the declaration of a non-generic type. In order to differentiate between them, the key word CONSTRUCTOR is put immediately after the key word TYPE in the declaration of a generic type. The different genericity parameters affecting types must also be indicated (see section 8.1).

In the same way, the declaration of a generic class resembles the declaration of a non-generic class. In order to differentiate between them, the key word CONSTRUCTOR is put immediately after the key word CLASS in the declaration of generic classes. The different genericity parameters affecting either the size constant (see section 8.2), or the types (see section 8.1) must also be indicated.

N.B. : the generic super-type (the super-class) of every type (every class) is always the type (the class) Top. Generic types and classes are not organised in a hierarchy.

8.1 Generic Type Parameters

8.1.1 Unrestricted Genericity

In this kind of genericity, the type of a generic formal parameter is not indicated and the type of the actual generic parameter is one of the types offered by the language.

Guide allows unrestricted genericity on the following types : REF aType, Integer, LongInt, ShortInt, Char, Boolean, but structured types may not be used as actual generic parameters.

In this form of genericity, a method call cannot be applied to generic type variables. (The method call is not defined on the type Integer and the type of the actual parameter is not known in advance, but operations defined on all types may be applied to these variables : assignment, equality, difference, comparison (:=, =, #, <, >, <=, >=).

At the declaration of a typed reference variable by a type constructor, the actual generic parameter must be specified. On the other hand, these are not recalled on creation of the instance.

Example

```
TYPE CONSTRUCTOR List OF [ T ] IS
  // T is the identifier to be substituted
  METHOD Insert (IN o : T);
  METHOD Delete : T;
  ...
END List.

CLASS CONSTRUCTOR List OF [ T ] IMPLEMENTS List IS
  // T is the identifier to be substituted
  CONST max : Integer = 100;
```

```
  list : Array[max] OF T;
  last : Integer = 0;
  METHOD Insert (IN o : T);
  BEGIN
    list[last] := o;    // o is a generic type variable
    last := last+1;
  END Insert;
  METHOD Delete : T;
  ...
END List.

/* example of use */
r : REF List OF Integer;
s : REF List OF REF Document;
r := List.New;
s := List.New;
```

8.1.2 Restricted Genericity

A restricted generic type could only be instantiated by a type conforming to the formal generic parameter. One of the consequences of this decision is that the methods defined for this type can be applied to the generic type variables. As it is not possible to define the sub-types for simple types, this form of genericity only applies to constructed types (the generic parameters are in the form REF aType).

Example

```
TYPE Document IS
name : String[20];
END Document.

TYPE CONSTRUCTOR Collection OF [ T : REF Document ] IS
  // T is the identifier to be substituted ;
  METHOD Search (IN title : String[20]) : T;
  METHOD Delete : T;
END Collection.

CLASS CONSTRUCTOR Collection OF [ T : REF Document ]
IMPLEMENTS Collection IS
  // T is the identifier to be substituted
  CONST max : Integer = 100;
  contents : Array[max] OF T;
  last : Integer = 0;
  METHOD Search (IN title : String[20]) : T;
  i : Integer;
  BEGIN
    i := 0;
    WHILE (title # contents[i].Title AND i<last) DO
```

```
/* when the type of the formal generic parameter is known, a
   method implemented on this type can be applied to the
   variable. Title is a method of the type document which
   returns the title of the document. */
      i := i+1;
    END;
    IF i # last THEN
      RETURN contents[i];
    END;
  END Search;

  METHOD Delete : T;
  ...
END Collection.

/* example of use */
r : REF Collection OF REF Report;
  /* Collection is a generic type which uses genericity
     restricted by the type Document. The type of the actual
     parameter may be any type which conforms to it, as in
     this example Report */

r := Collection.New;
  // creation of an object collection of reports.
```

Note : There is no relationship between genericity and conformance : for example : if the type Report conforms to the type Document, it is not possible to deduce from the conformance that the type List OF Report conforms to the type List OF Document.

8.2 Generic size parameters

The size parameter appears only in the declaration of the generic class and it is of the type Integer.

A generic class may implement either a non-generic type, in which case the only generic parameter is a size parameter, or a generic type (with generic parameters on the types).

Example

```
TYPE Array OF [ T ] IS
  METHOD Put (IN i : Integer; o : T);SIGNALS outside_the_limit;
  METHOD Get (IN i : Integer) : T; SIGNALS outside_the_limit;
END Array.

CLASS CONSTRUCTOR Array [size] OF [ T ] IMPLEMENTS Array IS
 // size is an identifier to be substituted ;
  contents : Array[size] OF T;
  METHOD Put (IN i : Integer; o : T);SIGNALS outside_the_limit;
```

```
  BEGIN
    IF (i < 0) AND (i >= size) THEN RAISE outside_the_limit;
    ELSE contents[i] := o ;
    END;
  END Put;
  METHOD Get (IN i : Integer) : T; SIGNALS outside_the_limit;
  BEGIN
    IF i < 0 AND i >= size THEN RAISE outside_the_limit;
    ELSE RETURN contents[i];
    END ;
  END Get;
END Array.

TYPE String IS
  METHOD Put (IN i :Integer;o :Char);SIGNALS outside_the_limit;
  METHOD Get (IN i : Integer) : Char;SIGNALS outside_the_limit;
END String.

CLASS CONSTRUCTOR String [size] IS
 // size is an identifier to be substituted ;
  contents : String[size];
  METHOD Put (IN i :Integer;o :Char);SIGNALS outside_the_limit;
  BEGIN
    IF i < 0 AND i >= size then RAISE outside_the_limit;
    ELSE contents[i] := o;
    END;
  END Put;
  METHOD Get (IN i : Integer) : Char;SIGNALS outside_the_limit;
  BEGIN
    IF i < 0 AND i >= size then RAISE outside_the_limit;
    ELSE RETURN contents[i];
    END;
  END Get;
END String.

/* example of use */
r : REF Array OF Integer;
r := Array[100].New;
s : REF String;
s := String[100].New;
```

8.3 Syntax

<generic_type_header> =
    TYPE CONSTRUCTOR <type_ident> OF
    '[' <generic_param> { ',' <generic_param> } ']' IS
<generic_param> =

```
        <restricted_generic_param> | <unrestricted_generic_param>
<restricted_generic_param> = <ident> ':' REF <constructed_type_ident>
<unrestricted_generic_param> = <ident>
<generic_class_header> =
        CLASS CONSTRUCTOR <class_ident> ['[' <dim_param> ']'] OF
        '[' <generic_param> { ',' <generic_param> } ']' IS
<dim_param> = <ident>
```

9 Conformance Rule

To ensure the validity of each method call on an object, the compiler makes a check on all types for the following instructions :

- assignment (and passing of parameters in procedures and methods),
- RETURN instruction.

Two kinds of checking, depending on the nature of the variables, are carried out by the compiler:

- for simple types, equality (or compatibility, cf Appendix B.1.6 and B.2.4) of the types is checked ;
- for assignments between reference variables, conformance is verified ;

In Guide, we have chosen not to restrict type checking of references to a comparative test in order to put to full use the possibilities of the object oriented model, particularly sub-typing: With a reference of type T, one wants to be able to designate all the objects of type T as well as those whose type is a sub-type of T, since, by definition, they offer at least the interface defined on the type T. More generally, the access to an object is authorised for all references whose type offers the access interface to that object. If this possesses other operations than those defined in the access reference type, they cannot be used with such a reference. Thus a mechanism for filtering access is obtained.

The expression of the syntactic conformance relationship is as follows (enriched to take account of the signals and parameters which are both arguments and results) :

Let two types be Tr and To ; To conforms to Tr, i.e. every syntactic check valid for Tr is valid for To, if and only if:

- for each operation defined by Tr, there is one (and only one) operation in To which conforms to it ;
- for each signal associated with the type Tr, there is a signal of the same name associated with the type To.

Let two operations be opo and opr ; opo conforms to opr if and only if :

- opo and opr have the same name, the same number of IN parameters, OUT parameters and IN_OUT parameters ;

- for each signal declared in To, there is a signal of the same name declared in Tr ;
- the type of the IN parameters of opr conforms to the type of the corresponding IN parameters of opo ;
- the type of the OUT parameters of opo conforms to the type of the corresponding OUT parameters of opr ;
- the type of the IN_OUT parameters of opo is equal to the type of the corresponding IN_OUT parameters in opr.

Example 1

Conformance comes from the sub-type relationship: Consider again the example given in section 5.4.3:

```
TYPE Account IS
  ...
END Account.

TYPE Savings_book_A SUBTYPE OF Account IS
  METHOD Calculate_interest (IN rate : Integer);
END Savings_book_A.

// Example of the use of conformance between the type Account
// and Savings_book_A

METHOD Print_balance (IN account : REF Account);
       // the method Print_balance accepts at input all
       // parameters of the type Account or conforming to Account
BEGIN
  output.WriteInteger(account.balance);
END Print_account;

// example of call
METHOD M ;
my_account : REF Account;
my_savings book_A : REF Savings_book_A;
BEGIN
  my_account : = Account.New;
  my_savings book_A := Savings_book_A.New;
  ...
  ref.Print_balance(my_account);
  ref.Print_balance(my_savings book_A);
   // correct call since the reference my_savings book_A is of
   // a type conforming to the type of the formal parameter
   // account (of the type Account) of the method M.
END M;
```

Example 2

Use of conformance in the case of a restriction relationship.

```
TYPE Library IS
  METHOD Borrow (IN doc :REF Document;IN who :REF Borrower;
                 OUT status : Integer);
  METHOD Return (IN doc : REF Document; IN who : REF Borrower;
                 OUT status : Integer);
  METHOD Add_doc (IN title, author, edition : String;
                   OUT doc : REF Document);
END Library.

CLASS Library IMPLEMENTS Library IS}
  // state of the method code
END Library.

TYPE User_library IS
  METHOD Borrow (IN book :REF Book; IN who :REF Borrower;
                 OUT status : Integer);
  METHOD Return (IN book : REF Book; IN who : REF Borrower;
                 OUT status : Integer);
END User_library.
```

(It is assumed that the type Book is a sub-type of the type Document.)

The type User_library is a restriction of the type Library : users are only allowed the operations borrow and return. In addition, the only documents which they may borrow are books. According to the definition of conformance, the type Library conforms to the type User_library. This gives access to the same library using different interfaces.

```
CLASS Administrator IMPLEMENTS Administrator IS
  lib : REF Library;
  // the administrator accesses the library with a reference
  // of the type Library (all operations are allowed)
  METHOD Init;
  command : String[15];
  BEGIN
    lib := ASSERTYPE(catal.Search("Guide_library");
    IF lib = NIL THEN
      lib := New.Library;
      catal.Insert("Guide_library", lib);
    END;
    REPEAT
      input.ReadString(command);
      CASE command OF
        "new_doc" : ...; lib.Add_doc (...); END;
        "consult_doc" : ... ; lib.Borrower(...); END;
        ...;
       END;
```

```
    UNTIL command = "terminated";
  END Init;
END Administrator.

CLASS User IMPLEMENTS User IS
  u_lib : REF User_library;
      // a user accesses the library with a reference
      // of the type User_library (restricted access)
  METHOD Init ;
  command : String[15];
  lib : REF Library;
  BEGIN
    lib := ASSERTYPE(catal.Search("Guide_Library");
    IF lib = NIL THEN RETURN;
    u_lib := lib;
           // valid allocation since the type of the reference
           // lib conforms to that of u_lib
    REPEAT
      input.ReadString(command);
        CASE command OF
          "consult_book" : ...; u_lib.Borrower(...); END;
          ...;
      END;
    UNTIL command = "terminated";;
  END Init;
END User.
```

Appendix A
Syntax of the Guide language

A.1 Type

```
<type> = <base_type> | <constructed_type>
<base_type> = <simple_type> | <structured_type>
<simple_type> = 'Integer' | 'ShortInt' | 'LongInt' | 'Char' | 'Boolean'
<structured_type> = record_type | array_type | string_type

<constructed_type> =
      (<type_header> | <non_constraint_generic_type_header>)
      { <synonyms> }
      { <visible_state_variable> }
      { <method_signature> }
      <end_type>

<type_header> =
      TYPE <constructed_type_ident> [ SUBTYPE OF <type_ident> ] IS
<generic_type_header> =
      TYPE CONSTRUCTOR <ident_type> OF
            '[' <generic_param> { ',' <generic_param> } ']' IS
<generic_param> =
      <constraint_generic_param> | <non_constraint_generic_param>
<constraint_generic_param> = <ident> : REF <ident>
<non_constraint_generic_param> = <ident>
<end_type> = END [ <constructed_type_ident> ]

<synonyms> =
      { SYNONYM <ident> '=' ( REF <constructed_type_ident>) | <base_type> ';' }

<visible_state_variable> =
      (<ident> { ',' <ident> } ':'
            <base_type> | REF <constructed_type_ident> [ '=' <constant_expr> ])
      | CONST <ident> { ',' <ident> } ':'
            (<base_type> | REF <constructed_type_ident>) '=' <constant_expr>

<method_signature> =
      METHOD <ident>
      [ '(' <param_list> ')' ]
      [ ':' <return_type> ] ';'
      [ <signal_list> ]

<param_list> = ( IN | OUT | IN_OUT ) <decl_param>
      { ';' ( IN | OUT | IN_OUT ) <decl_param> }
<param_decl> =
```

```
      [ <ident> { ',' <ident> } ':' ] (<base_type> | REF <constructed_type_ident>)
<return_type> = (<base_type> | REF <constructed_type_ident>)
<signal_list> = SIGNALS <ident> { ',' <ident> } ';'
```

A.2 Class

```
<class> =
      <class_header>
      [ <characteristics> ]
      { <synonyms> }
      { <state_variable> }
      { <method> | <procedure> }
      [ <exception_block> ]
      [ <restoration_block> ]
      [ <control> ]
      <end_class>
<class_header> =
      CLASS <class_ident> [ SUBCLASS OF <class_ident> ]
            IMPLEMENTS <constructed_type_ident> IS
<end_class> = END [<class_ident>] '.'

<generic_class_header> =
      CLASS CONSTRUCTOR <class_ident> ['[' <param_dim> ']']
      [ OF '[' <generic_param> { ',' <generic_param> } ']']
            IMPLEMENTS <constructed_type_ident> IS
<param_dim> = <ident>
<generic_param> =
      <non_contraint_generic_param> | <constraint_generic_param>
<non_contraint_generic_param> = <ident>
<constraint_generic_param> = <ident> ':' REF <ident_type_construit>

<characteristics> = ATTRIBUTE <characteristic> { ',' <characteristic> } ';'
<characteristic> = UNMOVABLE

<state_variable> =
      <ident> { ',' <ident> } ':'
            ( <base_type> | REF <constructed_type_ident> ) [ '=' <constant_expr> ]
      | CONST <ident> { ',' <ident> } ':'
            ( <base_type> | REF <constructed_type_ident>) '=' <constant_expr>

<method> =
      <method_header>
      { <work_variable> ';' }
      BEGIN
      <list_of_instructions>
```

```
        [ <exception_block> ]
        [ <restoration_block> ]
        END [<ident>] ';'

<method_header> =
        METHOD <ident>
        [ '(' <param_list> ')' ]
        [ ':' <return_type> ] ';'
        [ <signal_list> ]

<work_variable> = <state_variable>

<procedure> =
        <procedure_header>
        { <work_variable> ';' }
        BEGIN
        <list_of_instructions>
        [ <exception_block> ]
        [ <retoration_block> ]
        END [<ident>] ';'

<procedure_header> =
        PROCEDURE <ident>
        [ '(' <param_list> ')' ]
        [ ':' <return_type> ] ';'
        [ <signal_list> ]

<param_list> = ( IN | OUT | IN_OUT ) <param>
        { ';' ( IN | OUT | IN_OUT ) <param> }
<param> = <ident> { ',' <ident> }
                ':' (<base_type> | REF <constructed_type_ident>)
<return_type> = (<base_type> | REF <constructed_type_ident>)

<control> =
        CONTROL { <predefined_condition> | <non_predefined_condition> ';' }*

<non_predefined_condition> =
        <ident> { ',' <ident> }* ':' <boolean_expression>

<predefined_condition> =
        EXCLUSIVE '(' <method_list> ')'
        | WRITER'(' <method_list> '),' READER '(' <method_list> ')'
        | READER '(' <method_list> '),' WRITER'(' <methodes_list> ')'

<list_of_instructions> = <instruction> { ';' <instruction> }

<instruction> = ( <assignment>
```

```
      | <method_application>
      | <procedure_application>
      | <IF_instruction>
      | <CASE_instruction>
      | <TYPECASE_instruction>
      | <ASSERTYPE_instruction>
      | <FOR_instruction>
      | <WHILE_instruction>
      | <REPEAT_instruction>
      | <CO_BEGIN/CO_END_instruction>
      | <BLOCK_instruction>
      | <RETURN_instruction>
      | <RAISE_instruction>
      | BEGIN <list_of_instructions> [ <exception_block> ] END
      ) ';'

<assignment> = <variable> ':=' <expression>
<expression> = <ident> | <method_application> | <procedure_application>

<method_application> =
      <reference_variable>'.'<ident> [ '('<expression> { ',' <expression> } ')' ]

<procedure_application> =
      <ident> [ '('<expression> { ',' <expression } ')' ]

<IF_instruction> =
      IF <boolean_expression> THEN
      <list_of_instructions>
      [ ELSE <list_of_instructions> ]
      [ <exception_block> ]
      END

<CASE_instruction> =
      CASE <expression> OF
      { <list_of_labels> ':' <list_of_instructions> END ';' }*
      [ OTHERS ':' <list_of_instructions> ]
      [ <exception_block> ]
      END

<TYPECASE_instruction> =
      TYPECASE <reference_variable> OF
      { <constructed_type_ident> ':' <list_of_instructions> END ';' }*
      [ OTHERS ':' <list_of_instructions> ]
      [ <exception_block> ]
      END

<ASSERTYPE_instruction> =
```

```
      <reference_variable> ':=' "ASSERTYPE" '(' <expression> ')'

<FOR_instruction> =
      FOR <ident> ':=' <integer_expr> TO <integer_expr>
      [ BY <integer_expr> ] DO
      <list_of_instructions>
      [ <exception_block> ]
      END

<WHILE_instruction> =
      WHILE <boolean_expression> DO
      <list_of_instructions>
      [ <exception_block> ]
      END

<REPEAT_instruction> =
      REPEAT
      <list_of_instructions>
      [ <exception_block> ]
      UNTIL <boolean_expression>

<CO_BEGIN/CO_END_instruction> =
      CO_BEGIN
      { <ident> ':' <method_application> ';' }*
      [ <exception_block> ]
      CO_END [ <boolean_expression> | ALL | FIRST ]

<BLOCK_instruction> =
      <ident> ':=' BLOCK
            {<work_variable>}
            BEGIN
            <list_of_instructions>
            [<exception_block>]
            END

<RETURN_instruction> = RETURN [ <expression> ]

<RAISE_instruction> = RAISE <exception_ident> ';'

<exception_block> =
      EXCEPT
      { <exception_mask> ':' <traitant> }*
      END
<exception_mask> =
      ( <exception_ident> | ALL )
      [ FROM ( <constructed_type_ident> ["." <method_ident>] | SYSTEM | ALL ) ]
<traitant> = <instruction> ';'
```

Appendix B
Base Types in Guide

Base types can be simple or structured.

Objects which are of base type (called also base objects) may not be permanent. They are always created inside another object and are not managed by the system. It is prohibited to declare variables in the form x : REF Integer ; a base object is thus always internal to another object or local object. From base types (and classes), it is not possible to define other types (and classes) indirectly by use of the mechanisms of sub-typing and inheritance.

B.1 Simple types

An object in Guide is always composed of other objects with the exception of simple objects whose types are described below. The methods of simple types are defined by operators, simplifying the call syntax and making it similar to that of conventional programming. This syntax is only available for simple or predefined types and classes.

For example, one would write z := x + y and not z := x.+(y).

B.1.1 Boolean

```
TYPE Boolean IS
  PREFIX OPERATOR NOT : Boolean ;
  OPERATOR AND (IN Boolean) : Boolean ;
  OPERATOR OR  (IN Boolean) : Boolean ;
  OPERATOR XOR (IN Boolean) : Boolean ;
  OPERATOR = (IN Boolean) : Boolean ;
  OPERATOR # (IN Boolean) : Boolean ;
 END Boolean.
```

Two constant objects of the class Boolean are predefined ; their values are respectively 'true' and 'false'. The predefined symbolic names TRUE and FALSE allow the programmer to specify them.

B.1.2 Char

The type Char allows the declaration of character objects coded in 7 bit ASCII. A constant character is given between apostrophes.

```
TYPE Char IS
  OPERATOR = (IN Char) : Boolean ;
  OPERATOR # (IN Char) : Boolean ;
  OPERATOR < (IN Char) : Boolean ;
  OPERATOR > (IN Char) : Boolean ;
  OPERATOR <= (IN Char) : Boolean ;
  OPERATOR >= (IN Char) : Boolean ;
  OPERATOR + (IN Integer) : Char ; // successor in ASCII order
```

```
  OPERATOR - (IN Integer) : Char; // predecessor in ASCII order
     // these 2 operators return NIL outside the bounds
 END Char.
```

B.1.3 Integer

The type Integer allows the definition of integers coded in 32 bits.

```
TYPE Integer IS  }
  OPERATOR + (IN Integer) : Integer ; // addition
  OPERATOR - (IN Integer) : Integer ; // subtraction
  OPERATOR POWER (IN Integer) : Integer ; // power
  OPERATOR * (IN Integer) : Integer ; // multiplication
  OPERATOR / (IN Integer) : Integer ; // i.e. DIV
  OPERATOR DIV (IN Integer) : Integer ; // integer division
  OPERATOR MOD (IN Integer) : Integer;
               // remainder after integer division
  OPERATOR = (IN Integer) : Boolean ;
  OPERATOR # (IN Integer) : Boolean ;
  OPERATOR < (IN Integer) : Boolean ;
  OPERATOR > (IN Integer) : Boolean ;
  OPERATOR <= (IN Integer) : Boolean ;
  OPERATOR >= (IN Integer) : Boolean ;
  OPERATOR | (IN Integer) : Integer ;
               // operation 'or' bit by bit
  OPERATOR & (IN Integer) : Boolean ;
               // operation 'and' bit by bit
 END Integer.
```

B.1.4 ShortInt

This type has the same operators as the type Integer, but the size of its objects is 16 bits.

B.1.5 LongInt

This type has the same operators as the type Integer and the size of its objects is 32 bits.

B.1.6 Assignment rules between simple types

Type checking carried out by the compiler on simple types is based on type identity. However, the assignments defined in the table below are authorised.

On the other hand, assignments between Booleans and characters or integers are prohibited. Finally, it is possible to assign a character (or an integer) to an element of a string.

type of the variable	type of the expression	effect of assignment
Integer/LongInt/Short	Char	Assigns the ASCII code of the character to the integer variable.
Char	Integer/LongInt/ShortInt	The least significant 7 bits determines the ASCII code of the character variable.
Integer	ShortInt	Assigns the 16 bits of the ShortInt integer variable, with sign.
ShortInt	Integer	Truncates the most significant bits of the integer expression.
Integer/LongInt	LongInt/Integer	No problem because of same length

B.2 Structured types

B.2.1 Array[n] OF ...

The structured type Array allows the construction of multidimensional arrays of fixed size. The size parameters specify the number of elements of each dimension, the limits being always from 0 to size - 1. The declaration syntax is:

<array_type> =
'Array' '[' <integer_constant> { ',' <integer_constant> } ']' OF
(<base_type> | REF <constructed_type_ident>)

The defined operations are: the assignment of arrays and access to array elements by reading and writing with conventional syntax.

Example

```
tab : Array[20,20] OF Integer;
tab[5,5] := 10;   // write operation
elem := tab[5,5];  // read operation
```

B.2.2 Record

It is very easy to define record types and classes in Guide without additional syntax. Indeed, one can declare a type (and the corresponding class) consisting only of visible variables (not explicit methods). The only operations defined on objects of this type are reading and writing operations for these visible variables. This definition corresponds precisely with that of a conventional record structure.

However, as Guide does not allow the definition of types and classes within types and classes (i.e. there are no embedded types and classes), the type and class constructor Record is offered for the declaration of internal record objects. This constructor is implemented in this version by the structure Struct in C language.

The instantiation format of the constructor Record is that of Pascal.

<record_type> =
'Record'
{ <ident> ':' (<base_type> | REF <constructed_type_ident>) ';' }*
END

If one wishes to define a record structure internal to a class, in order to generate several instances of it, the structure must be declared with the clause SYNONYM.

Example

```
SYNONYM book =  Record
    title : String [15] ;
    author : String [10] ;
    contents : REF List OF REF Chapter;
 END ;
x, y : book;
z : Record
    a : Integer;
    b : String[10];
END;
```

Assignment Rule It is possible to assign one Record to another if the assignments type check field by field : i.e. there is conformance between reference type and constructed type fields, and type equality between simple type fields.

B.2.3 String[n]

The structured type String[n] allows the creation and use of strings of characters of fixed size. The parameter in the square brackets specifies the size of the string, but it is not obligatory in the formal parameters of the methods and procedures.

It is possible to access the characters of a variable of type String by specifying the order of the characters in the string. The bounds of a string are 0 and n-1. (The length is not coded in the structure.)

Constant character strings are given in inverted commas and must not be longer than the size of a line. An inverted comma in the middle of a string must be preceded by the character '\'.

The syntax of the type String :

<string_type> =
'String' ['['<constant_integer_expression> ']']
<string_constant> = ' " ' <list_of_characters> ' " '

The definition of the type constructor is :

```
TYPE CONSTRUCTOR String IS}
  OPERATOR = (IN String) : Boolean ;
  OPERATOR # (IN String) : Boolean ;
  OPERATOR < (IN String) : Boolean ;
  OPERATOR > (IN String) : Boolean ;
  OPERATOR <= (IN String) : Boolean ;
  OPERATOR >= (IN String) : Boolean ;
  OPERATOR +  (IN s1, s2 : String) : String;
      // string concatenation
 END String.
```

Example

```
string1 : String[20] ;
string2 : String[10] = "toto";
c : Char;
CONST string3 : String[4] = "tata";
c := string2[0]; // first character of string2 is  't';
METHOD Search (IN name : String) : REF Element;
```

In the current version of the compiler, this class is implemented with C character arrays.

B.2.4 Assignment rules between structured types

It is possible to assign arrays (of elements of the same type) or strings of different dimensions : if the assignment takes place in the direction from the largest to the smallest, there is truncation. A warning message is displayed by the compiler when the dimensions are not equal.

It is possible to assign an Array OF Char to a variable of the type String and vice versa.

Appendix C
Library of Guide Types

In this appendix, we present the different constructed types of Guide defined in the library which are currently available to the programmer. For each type we will give a description of the methods accessible to the user.

C.1 Block

The type Block, used in the instruction BLOCK (cf. 7.12) is defined as follows :

```
TYPE Block IS
  METHOD Value;
END Block.
```

C.2 Activity

```
TYPE Activity IS
  METHOD Kill;
  // destroy the activity
  METHOD Sleep (IN time : Integer);
  // suspend the activity for 'time' seconds
  METHOD Suspend;
  // stop the activity
  METHOD Resume;
  // awaken the activity
  METHOD Status : Boolean;
  // state of the activity. If TRUE, the activity is active ;
  // if FALSE, the activity is suspended or destroyed.
END Activity.
```

C.3 Domain

The type Domain allows certain primitives of the virtual machine to be made visible in a Guide program. In particular this allows the destruction of its own domain (terminating the execution of its application) by the instruction : MyDomain.Kill.

```
TYPE Domain IS
  chanIn : REF ChanIn;
  chanOut : REF ChanOut;

  METHOD Kill;
  // destruction of the domain
  METHOD Start(IN block : REF Block);
  /* creation of the initial activity of the domain. This has
     as its input/output channels the input/output channels
     of the activity which executes the method start. The
```

```
      principal activity created executes the block passed
      parametrically */
  METHOD Suspend;
  // suspension of a domain
  METHOD Resume;
  // resumption of a domain
  METHOD Status : Boolean;
  // state of the domain. If TRUE, the domain is active ;
  // otherwise FALSE
END Domain.
```

The predefined constant MyDomain contains a permanent reference of the type Domain.

C.4 Catal

The type Catal and the constant reference catal give access to the catalogue of objects. The function of this catalogue is to associate with each permanent object a symbolic name given by the programmer, with a life span independent of any programs.

```
TYPE Catal IS
  father : REF Catal = SELF;
  // give the containing directory ('..' in Unix)
  root : REF Catal = SELF;
  // give the root directory ('/' in Unix)
  nbEntries : Integer = 0;
  // give the number of objects in the directory

  METHOD ChangeOwner(IN newownername, localname : String) : Boolean;
  // to change the owner of an object or a directory

  METHOD CreateDir (IN localname, ownername : String;
                    OUT dir : REF Catal) : Boolean;
  // create a new directory in the current directory

  METHOD Delete(IN localname : String;
                OUT object : REF Top) : Boolean;
  METHOD Delete(IN localname : String; OUT object : REF Top);
  METHOD Delete(IN localname : String);
  // remove the association between an object and a symbolic name

  METHOD DeleteDir(IN localname : String;
                   OUT dir : REF Catal) : Boolean;
  /* remove the directory called "localname" from the current
     directory */

  METHOD Find(IN object : REF Top) : String;
  METHOD Find(IN object : REF Top; OUT name : String) : Boolean;
```

```
    METHOD RecursiveFind(IN object : REF Top;
                         pathname : String;
                         pathtable : Array[10] OF Catal): String;
    // find the symbolic name of the object whose reference is
    // passed parametrically

    METHOD Init(IN dad, slash : REF Catal);
    // initialise or modify the parent and the catalogue directory

    METHOD Insert(IN localname : String; object : REF Top);
    METHOD Insert(IN localname : String;
                  object : REF Top) : Boolean;
    METHOD Insert(IN localname : String; owner : String;
                  object : REF Top);
    METHOD Insert(IN localname : String; owner : String;
                  object : REF Top) : Boolean;
    // insert a new entry in the catalogue

    METHOD List;
    METHOD List(IN filter : String);
    // list the contents of the current directory
    // the filter allows a  sub-set to be selected from it

    METHOD SearchClass(IN localname : String) : REF Top;
    // retrieve the class reference whose symbolic name is
    // passed parametrically

    METHOD Search(IN localname : String) : REF Top;
    // retrieve the object reference whose symbolic name is
    // passed parametrically

    METHOD RecursiveSearch(IN localname, pathname : String ;
                           pathtable : Array[10] OF REF Catal);
    // list all the objects of a tree which have the same
    // symbolic name

    METHOD SearchDir(IN localname : String) : REF Catal;
    // retrieve the local directory reference whose
    // symbolic name is passed parametrically.

END Catal.
```

C.5 Chan

The type Chan gives access to the input/output system to programs written in Guide. The methods define the access inputs/outputs of the simple types. Following the implementation of the type Chan chosen, some of the following methods have no effect.

```
TYPE Chan IS
  /* The initialisation methods return the signal NOT_EXIST, if the
     file does not exist */
  METHOD Init (IN file : String);
  // open the file specified in read/write
  METHOD Init(IN file : String; beginRead, beginWrite : Boolean);
  /* open the file specified according to the following mode :
     if beginRead = TRUE, the read pointer is placed
     at the beginning. If not, it is placed at the end.
     if beginWrite = TRUE, the write pointer is placed at the
     beginning. If not, it is placed at the end. */

  /* all the read methods return the signal EOF, if
     the end of the file is reached */
  METHOD ReadInteger : Integer;
  // read the next integer found in decimal form
  METHOD ReadString : String;
  /* read the next character string (up to the character
    '\n'). This is not part of the result. */
  METHOD ReadChar : Char;
  // read the next character found
  METHOD ReadBool : Boolean;
  // read the next Boolean found
  /* TRUE is returned if the character string found equals "TRUE"
     or "true" ;  FALSE if not */

  METHOD Sync;
  // clear the input/output buffers
  METHOD Clear;
  // clear the screen
  METHOD Goto ( IN Integer; Integer );
  /* place the cursor on line i and column j */

  METHOD WriteRef( IN r : REF Top );
  // write a reference in the form : <container number>.<oid>
  METHOD WriteInteger( IN i : Integer );
  // write an integer in decimal form
  METHOD WriteString( IN s : String );
  // write the character string
  METHOD WriteChar( IN c : Char );
  // write a character
  METHOD WriteBool( IN b : Boolean );
  // write a Boolean.
 END Chan.
```

C.6 ChanIn

The type ChanIn is a restriction of the type Chan, consisting only of the input methods. The environment variable input is declared to be of this type.

```
TYPE ChanIn IS
  /* all read methods return the signal EOF, if the end
     of the file is reached */
  METHOD ReadInteger : Integer;
  // read the next integer found in decimal form.
  METHOD ReadString : String;
  /* read the next character string (up to the character
    '\n'). This is not part of the result. */
  METHOD ReadChar : Char;
  // read the next character found
  METHOD ReadBool : Boolean;
  // read the next Boolean found.
  // TRUE is returned if the character string found equals
  //   "TRUE" or "true" ; FALSE if not.
END ChanIn.
```

C.7 ChanOut

The type ChanOut is a restriction of the type Chan, consisting only of output methods. The environment variable output is declared of this type.

```
TYPE ChanOut IS
  METHOD Sync;
  // clear the input/output buffers  METHOD Clear;
  // clear the screen
  METHOD Goto ( IN Integer; Integer );
  /* place the cursor on line i and column j */
  METHOD WriteRef( IN r : REF Top );

  // write a reference in the form : <container number>.<oid>
  METHOD WriteInteger( IN i : Integer );
  // write an integer in decimal form
  METHOD WriteString( IN s : String );
  // write the character string
  METHOD WriteChar( IN c : Char );
  // write a character
  METHOD WriteBool( IN b : Boolean );
  // write a Boolean.
END ChanOut.
```

C.8 Class

```
TYPE Class IS
  /* all methods return the signal NOT_EXIST if the class
```

```
specified does not exist */

  METHOD New : REF Top;
  /* create an object of the class specified at the site
     of execution of the activity */
  METHOD New (IN site : String) : REF Top;
  /* create an object of the class specified at the site
     specified */
  METHOD New (IN site : String; size : Integer) : REF Top;
  /* create an object of the class specified at the site
     specified and of the size given (used essentially by the
     compiler). If the size specified is less than the size of an
     instance of the class take the size of the class */
  METHOD New (IN site : String; size : Integer;
              flag : Integer) : REF Top;
  /* create an object of the class specified at the site
     specified, of the size given (used essentially by the
     compiler) with the characteristics specified (cf 6.1)).
     If the size specified is less than the size of an
     instance of the class take the size of the class */
  METHOD NewOnSite (IN site : String) : REF Top;
  /* create an object of the class specified at the site
     specified */
  METHOD NewWithSize (IN size : Integer) : REF Top;
  /* create an object of the class specified of the
     size given (used essentially by the compiler).
     If the size specified is less than the size of
     an instance of the class, take the size of the
     class */
  METHOD NewWithFlag (IN flag : Integer) : REF Top;
  /* create an object of the class specified with the
     characteristics specified */
END Class.
```

C.9 Top

The type Top is the super-type of all the others.

```
TYPE Top IS
  SYNONYM Predicate = Record
    caller : Top ;
    method : String ;
  END;

  METHOD InitOnNew;
  METHOD GetRef (OUT nb : Integer;
                     listref : Array OF REF Top);
    // return the number and the list of references appearing
```

```
    // in the state of the object
  METHOD PrintState;
    // display the state of the object
  METHOD PrintCounter;
    // display the value of the synchronisation counters
    // associated with the object
  METHOD ObjectSize : Integer;
  METHOD ObjectTypeName : String;
  METHOD ObjectClassName : String;
  METHOD ObjectTypeRef : REF Top;
  METHOD ObjectClassRef : REF Top;
  METHOD Destroy;
END Top.
```

Appendix D
Compilation and execution

D.1 Compilation

A Guide application is made up of an assembly of interacting objects. The construction of an application corresponds to the definition of the types and classes of these objects as well as to the designation of the object and the initiating method of the application. For example, a mail application could create objects of the classes MailUserInterface, Mailbox and Mail and a session could be started by calling the method InitSession on an object of the class MailUserInterface.

The types and classes to be compiled must be given to the compiler in the order in which they are used. A file to be compiled may contain as many types and/or classes as required.

The different files generated at compilation are saved by the compiler in the directory .GUIDE/name_user/name_type (or name_class). The directory .GUIDE is found in the directory defined by the environment variable $GUIDESYS (in general /usr/local/GUIDE).

In what follows, we describe briefly how to use the GUIDE compiler. The reader will find a detailed description of the functioning of the compiler in the User Manual of the compiler [4].

D.1.1 Principles of type compilation

A type is compiled in three passes. The first consists of a lexical and syntactic analysis of the source. The second verifies the description of the type according to its place in the hierarchy of types (respecting the rule of over-riding in the case of a sub-type). At the conclusion of this pass a description file of the type is generated and is stored in the file type_name/TYPE.

In order to compile a type it is sufficient to call the Guide compiler as follows: glc file_name where file_name is the name of the file which contains the source code of the type (or types) to be compiled.

D.1.2 Principles of class compilation

A class is compiled in five passes. The first consists of a lexical and syntactical analysis of the source. The second checks whether the programming rules of Guide are respected (rule of construction of a sub- class, rule of conformance). The third produces a set of files which contain C code - the description of the class, a description of its state and the code of its methods or procedures. The fourth pass is the compilation of this C code and the fifth the loading of the class compiled into the Stable Store (MPO).

The main options are :

glc file_name // compilation up to pass 3 (production of C code).

glc -p5 file_name // total compilation (loading the class into MPO).

D.1.3 Options of the Guide Compiler

Invoking the compiler : glc [option] file

The command glc without argument produces the list of compilation options.

Options common to the compilation of types and classes

- -p1 the compiler only carries out pass 1 (lexical analysis) on a file containing a sequence of type and class descriptions. The source file is sought relative to the current directory. No file is generated.
- -p2 : the compiler carries out passes 1 and 2 (syntactical analysis and verification of conformance) on a file containing a list of type and class descriptions. The source file is sought relative to the current directory. For each type, a file containing its description is generated called .T/aType.TYPE. For classes, the file is called: .C/aClass.CLASS.

Options for the compilation of classes

- -p3 the compiler carries out passes 1, 2 and 3 (generation of C code) on a file containing a list of type and class descriptions. The source file is sought relative to the current directory.

  For each class compiled, a set of files is

  generated :

  1. .GUIDE/name_class/header.c : contains inclusion directives for all *.h files used (including system files). In addition, it contains the description of the execution structure of the class.
  2. .GUIDE/name_class/D : contains the description of the state of the objects of the class.
  3. .GUIDE/name_class/M : contains the code of the methods defined in the class.

- -p4 : the compiler carries out passes 1, 2, 3 and 4 (compilation of C code and resolution of external references) on a file containing a sequence of type and class descriptions. The source file is sought relative to the current directory. For each class, the generated file is called : .GUIDE/class_name/g.
- -p5 : the compiler carries out passes 1, 2, 3 ,4 and 5 (loading of the class in the MPO and its insertion into the class catalogue) on a file containing a sequence of type and class descriptions. The source file is sought relative to the current directory. For each class, the generated file, which is called : .GUIDE/class_name/e, is a copy of that inserted in the MPO.
- -r : the compiler keeps the intermediate files.
- -F : the compiler forces the compilation of types and classes (even if they have been compiled before).

By default the compiler is called with the option -p3.

D.2 Execution

An application is initiated by designating a class. The system creates the domain corresponding to the application, creates within that domain an object of the designated class, then creates the principal activity of the domain by giving it the job of executing the method Init defined on the first object of the domain. The types and classes necessary to the application must be compiled before this execution.

In the current version, the method to execute at the initiating of an application is a method without parameters.

The activity allowing the initiation of an execution is called : **guide**

This activity allows the following commands :

- ls : list the classes and objects present in the catalogue,
- class_name : create an instance object of this class and activate the method Init.
- class_name.method_name : create an instance object of this class and activate the method method_name.

References

[1] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina, "Design and Implementation of an object-oriented, strongly typed language for distributed applications" *JOOP,* vol. 3, num. 3, pp. 11-22, September/October 1990.

[2] S. Lacourte, "Note 61 : Le modele d'exception de Guide et son implementation" *Note interne Guide,* num. 61, Juillet 1990.

[3] H. Nguyen Van, *Compilation et environnement d'execution d'un langage a base d'objets,* These, Institut National Polytechnique de Grenoble, Janvier 1991.

[4] M. Riveill, "Note 53 : Manuel utilisateur du compilateur Guide" *Note interne Guide,* num. 53, Janvier 1990.

Contents

1 Introduction **1**

2 Programming Rules **2**
2.1 Character Set . . . 2
2.2 Identifiers . . . 2
2.3 Key Words . . . 3
2.4 Symbols . . . 3
2.5 Reserved Words . . . 3
2.6 Comments . . . 4

3 Guide Program **4**

4 Objects, variables and references **5**
4.1 Definitions . . . 5
4.2 Scope of a variable . . . 6
4.3 Objects and references . . . 7
4.3.1 Creation of an object . . . 7
4.3.2 Explicit destruction of an object . . . 7
4.3.3 Designation of the current object : SELF . . . 8
4.3.4 Constant reference NIL . . . 8
4.4 Environment Variables . . . 8

5 Constructed Types **9**
5.1 Synonyms . . . 9
5.2 Visible state variables . . . 10
5.2.1 Declaration and initialisation . . . 10
5.2.2 Constant visible state variable . . . 11
5.2.3 Syntax . . . 11
5.2.4 Restrictions . . . 11
5.3 Methods . . . 11
5.4 Construction of a sub-type . . . 13
5.4.1 Overriding of visible state variables . . . 13
5.4.2 Signature overriding . . . 13
5.4.3 Example . . . 13

6 Constructed Classes **14**
6.1 Characteristics of the objects of a class . . . 16
6.2 Synonyms . . . 16
6.3 State Variables . . . 16
6.3.1 Initialised State Variables . . . 17
6.3.2 Constant State Variables . . . 17
6.3.3 Syntax . . . 18
6.4 Methods . . . 18
6.5 Procedures . . . 19
6.6 Paramater Passing . . . 19
6.6.1 IN Parameters . . . 19

6.6.2 OUT Parameters 20
6.6.3 IN_OUT Parameters 20
6.6.4 Return Parameter 20
6.7 Activation Conditions 21
6.8 Construction of a sub-class 23
6.8.1 Over-riding of the state model 23
6.8.2 Over-riding of a signature 24
6.8.3 Code over-riding 24
6.8.4 Over-riding of the control clause 24
6.8.5 Over-riding of the blocks EXCEPT and RESTORE 25

7 Instructions **25**
7.1 Assignment and expressions 26
7.2 Method Call 26
7.3 Procedure Call 27
7.4 Instruction IF 27
7.5 Instruction CASE 27
7.6 Instruction TYPECASE 28
7.7 Instruction ASSERTYPE 29
7.8 Instruction FOR 30
7.9 Instruction WHILE 30
7.10 Instruction REPEAT 30
7.11 CO_BEGIN Instruction 31
7.12 BLOCK Instruction 32
7.13 Instruction RETURN 34
7.14 Instructions relating to exceptions 34
7.14.1 Description of the exception mechanism 34
7.14.2 Specific Instructions 36
7.14.3 Exceptions in a handler 37
7.14.4 Specific Cases of Signalling 37
7.14.5 Exceptions and CO_BEGIN 37
7.14.6 Restoration 38
7.14.7 Interruptions 38
7.14.8 System Exceptions 39
7.14.9 Syntax 39
7.14.10 Example 39

8 Genericity **40**
8.1 Generic Type Parameters 41
8.1.1 Unrestricted Genericity 41
8.1.2 Restricted Genericity 42
8.2 Generic size parameters 43
8.3 Syntax 44

9 Conformance Rule **45**

Appendix A Syntax of the Guide Language **49**
A.1 Type . . . 49
A.2 Class . . . 50

Appendix B Base Types in Guide **54**
B.1 Simple types . . . 54
B.1.1 Boolean . . . 54
B.1.2 Char . . . 54
B.1.3 Integer . . . 55
B.1.4 ShortInt . . . 55
B.1.5 LongInt . . . 55
B.1.6 Assignment rules between simple types . . . 55
B.2 Structured types . . . 56
B.2.1 Array[n] OF 56
B.2.2 Record . . . 56
B.2.3 String[n] . . . 57
B.2.4 Assignment rules between structured types . . . 58

Appendix C Library of Guide Types **59**
C.1 Block . . . 59
C.2 Activity . . . 59
C.3 Domain . . . 59
C.4 Catal . . . 60
C.5 Chan . . . 61
C.6 ChanIn . . . 63
C.7 ChanOut . . . 63
C.8 Class . . . 63
C.9 Top . . . 64

Appendix D Compilation and execution **66**
D.1 Compilation . . . 66
D.1.1 Principles of type compilation . . . 66
D.1.2 Principles of class compilation . . . 66
D.1.3 Options of the Guide Compiler . . . 67
D.2 Execution . . . 68