

IDL/LISP MAPPING

VERSION 1.0

Franz Inc.

Copyright 1998 by Franz Inc.

Copyright © 1998 by FRANZ INC. All rights reserved. This is revision 0.6 of this document..

ESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013.

CORBA and Object Request Broker are trademarks of Object Management Group.

OMG is a trademark of Object Management Group.

DRAFT PROPRIETARY TO FRANZ INC.

Table of Contents

1 Preface	9
1.1 Status	9
1.2 Scope	9
1.3 Intended audience.....	10
1.4 Missing Items	10
1.5 Conventions	10
1.6 Version of Lisp.....	10
1.7 Contact Points	10
1.8 Acknowledgments	10
2 Mapping and IDL	13
2.1 Introduction to IDL.....	13
2.2 How IDL is used.....	13
2.3 Mapping constituents	14
2.3.1 Mapping the primitive data types.....	14
2.3.2 Mapping the constructed data types.....	14
2.3.3 Interfaces	14
2.3.4 Mapping the syntax.....	15
2.3.5 Mapping the names	15
2.3.6 Mapping pseudo-interfaces	15
2.4 Mapping summary	15
3 Mapping IDL to Lisp	17
3.1 Mapping concepts.....	17
3.2 Semantics of type mapping	18

3.3	Mapping for basic types	18
3.3.1	Overview	18
3.3.2	boolean	20
3.3.3	char	20
3.3.4	octet	20
3.3.5	wchar, wstring	21
3.3.6	string	21
3.3.7	Integer types	21
3.3.8	Floating point types	21
3.3.9	fixed	21
3.4	Introduction to named types	22
3.4.1	Naming terminology	22
3.5	Distinguished packages	23
3.5.1	Nicknames for distinguished packages	24
3.6	Scoped names and scoped symbols	24
3.6.1	Definitions	24
3.7	The <code>package_prefix</code> pragma	25
3.8	Mapping for interface	26
3.8.1	Example	27
3.9	Mapping for operation	27
3.9.1	Parameter passing modes	27
3.9.2	Return values	27
3.9.3	one-way	28
3.9.4	Efficiency optimization: using macros instead of functions	28
3.9.5	exception	28
3.9.6	context	28
3.9.7	Example	29
3.10	Mapping for attribute	29
3.10.1	readonly attribute	30
3.10.2	normal attribute	30
3.10.3	Example	30
3.11	Mapping of module	30
3.11.1	Example	30
3.12	Mapping for enum	31
3.12.1	Example	32
3.13	Mapping for struct	32
3.13.1	Example	33
3.14	Mapping for union	33
3.14.1	Member accessors	34
3.14.2	Example	34
3.15	Mapping for const	35
3.15.1	Example	35
3.16	Mapping for array	36

3.16.1 Example	36
3.17 Mapping for sequence	36
3.17.1 Example	38
3.18 Mapping for exception	38
3.18.1 User exception	39
3.18.2 System exception	39
3.19 Mapping for typedef	39
3.19.1 Example	40
3.20 Mapping for any	40
3.20.1 Constructors	40
3.20.2 Typecode accessor	40
3.20.3 value accessor	41
3.20.4 Interaction with GIOP	41
3.20.5 Additional examples of any usage	42
3.21 Mapping Overview	42
3.21.1 Rule 1: How names of types are formed	43
3.21.2 Rule 2: How names of operations are formed	43
3.21.3 Rule 3: Lisp functions corresponding to IDL types	43
4 Mapping Pseudo-Objects to Lisp	45
4.1 Introduction	45
4.2 Certain exceptions	45
4.3 Environment	46
4.4 NamedValue	46
4.5 NVList	46
4.6 Context	47
4.7 Request	47
4.8 ServerRequest	48
4.9 TypeCode	48
4.10 ORB	49
4.10.1 ORB initialization	49
4.10.2 ORB pseudo-object	49
4.11 Object	51
4.12	Principal 51
4.13 DynAny	51
4.14 The IDL Compiler	52
5 Server-Side	53
5.1 Introduction	53
5.2 Mapping of native types	53
5.3 Implementation objects	53

5.4	Servant classes	53
5.4.1	Note on proxies	54
5.5	Defining methods	54
5.5.1	Syntax of corba:define-method	54
5.5.2	Description	54
5.6	Examples	55
5.6.1	Example: A Named Grid	55

6 Design Decisions59

6.1	Introduction	59
6.1.1	Goals.	59
6.1.2	Lisp Version	60
6.1.3	Reverse mapping	61
6.1.4	Compiler interface	61
6.1.5	Type checking	61
6.2	Overall Design Philosophy.	61
6.2.1	Relationship to other mappings	62
6.3	Names	62
6.3.1	Capitalization	62
6.3.2	Nesting	62
6.3.3	Character set.	63
6.3.4	Alternative mappings	63
6.3.5	Prefixes.	63
6.4	Mapping of basic types.	64
6.4.1	boolean	64
6.4.2	float and double	64
6.5	Mapping for struct	64
6.6	Mapping for exception	64
6.6.1	condition hierarchy	64
6.6.2	Naming exception classes	65
6.7	Mapping for enum	66
6.8	Mapping for union	66
6.9	Mapping of module	66
6.10	Mapping for array	66
6.11	Mapping for sequence	67
6.11.1	Advantages of our proposal	68
6.11.2	Disadvantages of our proposal.	68
6.11.3	Conclusion	68
6.12	Mapping for any	68
6.13	Mapping for typedef	69
6.14	Mapping for interface	69
6.15	Mapping for operation : the name	69

6.15.1	Explicit operation mapping	70
6.15.2	Use of a designated package	71
6.15.3	Using a prefix	72
6.15.4	Using the :keyword package	72
6.15.5	Conclusion	73
6.16	operation mapping: signature	73
6.16.1	Leave the signature of the generic function unspecified	73
6.16.2	Require method definition via a particular macro	73
6.17	operation mapping: parameter passing modes	74
6.18	Mapping of attribute	74
6.19	Compiler mapping	75
6.20	Pseudo Interface Mapping	75
6.21	Server side mapping	75

1.1 Status

This document presents a proposed IDL/Lisp language mapping. It is being circulated for review to interested members of the Lisp/CORBA community.

Because this document is in preliminary form, it contains a number of formatting and editing problems:

- Some mapping features are not illustrated with examples.
- The “rationale” section may not be up-to-date with respect to the actual mapping.
- Many of the fonts are not used properly, and the document formatting has various errors.
- Some of the examples may be out-of-synch with the current version of the mapping.
- Some of the explanations are terse to the point of being elliptical.

If the Lisp community concurs with the main ideas presented in the mapping, the document will be edited and formatted to professional standards and submitted to the OMG. The problems discussed above will be corrected, the design rationale section will be shortened and placed at the beginning (although a separate document including detailed design rationale decisions will be made available) and unclear semantic issues will be clarified. Furthermore, appropriate front matter, such as acknowledgments and copyright clearances will be included.

1.2 Scope

This document is intended only to deal with matters concerning the IDL/Lisp language mapping. In particular, there are few explanatory examples and matters of launching and use of the services are not discussed.

1.3 *Intended audience*

This document is intended for readers who are familiar with both IDL and with Lisp. However, Chapter 2, "Mapping and IDL", contains a brief introduction to certain mapping concepts, but this chapter will not be included in the version submitted to the OMG.

1.4 *Missing Items*

The following topics are incompletely specified or not specified at all

- Portable Object Adaptor
- possible new PIDL and other required mappings
- **DynAny** type management

1.5 *Conventions*

IDL appears using this font.

Lisp code appears using this font.

(This usage is inconsistent in this version of the document).

1.6 *Version of Lisp*

This document is based on Common Lisp specified in X3J13 Committee, *ANSI X3.226-1994, American National Standard for Information Technology—Programming Language—Common Lisp*, ANSI (New York) 1994

1.7 *Contact Points*

Questions and comments about this document are encouraged and should be directed to:

Lewis Stiller
Franz Inc.
1995 University Avenue
Berkeley, CA 94704
phone: 510-548-3600
fax: 510-548-8253
email: stiller@franz.com or orblink@franz.com

1.8 *Acknowledgments*

The design of this mapping was influenced by a number of sources outside of Franz Inc.

We used the ILU system and its mapping both for design guidance and for assessing practical experience. We thank Bill Janssen of Xerox Parc for providing us with access to ILU and for explicating the design decisions in the mapping used by ILU. We thank Joachim Achtzehnter for his work on the design of ILU and for his help in preparing this mapping document.

We would like to thank Ken Anderson of BBN for his comments on suggestions on this mapping.

We would like to thank Greg Whittaker of Mitre Corporation for his comments and suggestions on this mapping.

We would like to thank Stanley Knutson of Concentra for his comments.

We also used a mapping due to Thomas Mowbray of Mitre Corporation.

We are grateful for the assistance of Harlequin Inc. in preparing this mapping.

This chapter briefly reviews some concepts of IDL and defines the notion of a language mapping. A summary of the IDL/Lisp mapping is presented. [This chapter will not be included in the version of this document that will be submitted to the OMG].

2.1 Introduction to IDL

IDL, or Interface Definition Language, is a language defined by the Object Management Group.

The key data type in IDL is the interface, which describes the behavior of an objects that implements that interface. The IDL definition for an interface describes all of the operations to which an object that implements that interface can respond. For each such operation, it describes the allowed types of the parameters to the operation and the allowed type of the value returned by the operation.

IDL allows the types other than interfaces to be expressed. For example, primitive types such as boolean, several signed and unsigned integral types, and some floating point types may be defined.

Constructed types analogous to the C struct or Pascal record type may be defined, and some simple type aliasing is possible in a way analogous to the C typedef construct.

Arrays and sequences may also be defined.

2.2 How IDL is used

IDL is typically used in the following manner. An server process wishes to make some of its functionality available for invocation by clients. These clients may not be in the same process, on the same machine, or even written in the same language.

The server publishes the IDL definitions that define the interfaces of the objects that it implements. A client can use those definitions to invoke operations on objects that reside within the server process.

The syntax used by the client to invoke a method on an object defined in IDL, and the relationship between the data types specified in IDL and the native data types of the language in which the client is implemented is defined by the mapping of IDL into that language.

This document describes a mapping from IDL into Common Lisp.

2.3 *Mapping constituents*

Informally speaking we can divide a mapping into these categories.

2.3.1 *Mapping the primitive data types.*

IDL implicitly assumes that there is a universe of primitive data values, certain sets of which may be denoted by IDL types.

The mapping will, for each abstract IDL data value define the associated Lisp data value. The set of IDL data values corresponding to a particular IDL data type will correspond to the Lisp type whose elements are the Lisp values that correspond to each IDL value in that set.

For example, IDL has a concept of the integer constant 12. It seems reasonable that this value would correspond to the Lisp value 12, and indeed, in our mapping, it does.

In fact, each IDL integer value corresponds to precisely one Lisp integer of the same value.

One of IDL's predefined types is **unsigned short**, which comprises the set of values between 0 and 65535 inclusive. The Lisp type corresponding to this IDL type is thus the set of Lisp integers between 0 and 65535, a set specified in Lisp by the type specifier (**integer 0 65535**) or, equivalently, (**unsigned-byte 16**).

The primitive data types are **boolean, double, long double, float, octet, short, unsigned short, long, unsigned long, long long, unsigned long long, char, string, any**.

2.3.2 *Mapping the constructed data types*

The constructed data types are union, struct, array, exception, and sequence. These correspond to aggregates or collections of other IDL elements. In each case we need to determine whether such a type maps most naturally to an instance of standard-class, an instance of structure-class, or to some other Lisp construct.

2.3.3 *Interfaces*

The most important data type to map is the interface data type.

2.3.4 *Mapping the syntax.*

How are methods on objects invoked? How are methods defined?

For example, in Lisp we would ask: does method invocation correspond to function invocation, generic function invocation, or macro invocation? Are methods defined using `defun`, `defmethod`, or some other syntax?

2.3.5 *Mapping the names*

It is necessary to assign a Lisp symbol that represents each named IDL construct. What symbol should correspond to a given operation, or a given interface? How are capitalization and package-names handled?

2.3.6 *Mapping pseudo-interfaces*

IDL has certain constructs that behave like interfaces in some ways but that are not full fledged interfaces. For example, ORB, the interface that describes the Object Request Broker itself, is a pseudo-interface. These are typically mapped separately.

2.4 *Mapping summary*

Most of the material in this mapping document concerns fairly esoteric issues that rarely arise in practice. The main points of our mapping are as follows.

Primitive data types are mapped to corresponding primitive data types in Lisp.

struct and union are mapped to classes. Each member of the struct or union can be accessed using a regular syntax.

Arrays map to arrays.

Sequences can map either to lists or to vectors; that is, sequences map to sequences.

Exceptions are mapped to conditions.

Interfaces are mapped to classes, and interfaces that inherit map to classes that inherit.

Operations on interfaces map to methods on a generic function. This generic function discriminates only on its first argument, which is interpreted as the receiver of the operation.

The module in which an IDL entity is declared is mapped to the package name of the corresponding symbol. The name of the symbol is formed from the rest of the scope of the module.

A mapping to the IDL compiler is included.

This section describes the mapping of IDL into the Lisp language.

The rationale for design decisions can be found in Chapter 6, “Overall Design Rationale”.

In most cases examples of the mapping are provided. It should be noted that the examples are code fragments that try to illustrate only the language construct being described.

3.1 Mapping concepts

By an *IDL entity* we mean an element defined in some IDL file.

For example, consider the code fragment

```
module A {  
    interface B {  
        void op1(in long bar);  
    };  
}
```

The IDL entities are the module named “**A**”, the interface named “**B**”, the operation named “**op1**”, the formal parameter named “**bar**”, and the primitive data types **void** and **long**.

Our mapping will associate to each IDL entity declared in an IDL specification a corresponding Lisp entity.

The Lisp entity corresponding to a given IDL entity will be said to be *generated* from the IDL entity.

If the IDL entity has a name then the corresponding Lisp entity will also have a name. Whereas IDL entities are named by strings (i.e., identifiers), Lisp entities are named by symbols.

It is the goal of this chapter to specify, for each IDL construct, the Lisp entity, and the name of that entity, that is generated by the mapping.

3.2 *Semantics of type mapping*

The statement that an IDL type I is mapped to a Lisp type L indicates if V is a Lisp value whose corresponding IDL type I , then the consequences are not specified if the value of V is not a member of the type L .

For example, if V is passed as an parameter to an IDL operation or if V is returned from an IDL operation, then a conforming implementation may reasonably perform any of the following actions if V is not of the type L .

- If V may be coerced to L , then V may be replaced by the result of coercing V to the type L .
- If V cannot be coerced to L , then an error may be signalled. If the error occurs during marshalling or unmarshalling, **corba:marshal** should be signaled.

3.3 *Mapping for basic types*

3.3.1 *Overview*

The following table shows the basic mapping.

The first column contains the IDL name of the IDL type to be mapped. Each IDL type denotes a set of IDL abstract values.

The set of values denoted by an entry in the first column will be mapped under the mapping described in this document to a set of Lisp values. That set of Lisp values is described in two ways:

- The entry “Name of Lisp type” is a symbol that names the type represented by this set of Lisp values.

- The entry “Lisp type specifier” is a standard Common Lisp type specifier that denotes this set of Lisp values.

Figure 3-1 BASIC TYPE MAPPINGS

IDL Type	Name of Lisp type	Lisp type specifier
boolean	corba:boolean	boolean
char	corba:char	character
wchar	corba:wchar	see text
octet	corba:octet	(unsigned-byte 8)
string	corba:string	string
wstring	corba:wstring	see text
short	corba:short	(signed-byte 16)
unsigned short	corba:ushort	(unsigned-byte 16)
long	corba:long	(signed-byte 32)
unsigned long	corba:ulong	(unsigned-byte 32)
long long	corba:longlong	(signed-byte 64)
unsigned long long	corba:ulonglong	(unsigned-byte 64)
float	corba:float	see text
double	corba:double	see text
fixed	corba:fixed	see text

Additional details are described in the sections following.

3.3.1.1 Example

```
(typep -3 'corba:short)
> T
```

```
(typep -3 'corba:ushort)
> nil
```

```
(typep "A string" 'corba:string)
> T
```

3.3.2 boolean

The IDL boolean constants **TRUE** and **FALSE** are mapped to the corresponding Lisp boolean literals **T** and **nil**. The type specifier **corba:boolean** specifies this type.

3.3.3 char

IDL **char** maps to the Lisp type **character**. The type specifier **corba:char** specifies this type.

3.3.3.1 Usage example

```
(typep #\x 'corba:char)
> T
```

```
(typep "x" 'corba:char)
> nil
```

3.3.4 octet

The IDL type **octet**, an 8-bit quantity, is mapped as an unsigned quantity to the type **corba:octet**. The type specifier **corba:octet** denotes the set of integers between **0** and **255** inclusive. This set can also be denoted by the type specifier (**unsigned-byte 8**).

3.3.4.1 Usage example

```
(typep 255 'corba:octet)
> T
(typep -1 'corba:octet)
> nil
```

3.3.5 wchar, wstring

The types **wchar** and **wstring** are mapped to Lisp types named **corba:wchar** and **corba:wstring**. The type **corba:wstring** must be a subtype of **corba:sequence** whose constituents can elements of type **corba:wchar**.

3.3.6 string

The IDL **string**, both bounded and unbounded variants, are mapped to **string**. Range checking for characters in the **string** as well as bounds checking of the **string** shall be done at marshal time. The type specifier **corba:string** denotes the set of Lisp **strings**.

3.3.6.1 Usage example

```
(typep "A string" 'corba:string)
> T
(typep nil 'corba:string)
> nil
```

3.3.7 Integer types

The integer types each map to the Lisp **integer** type. Each IDL integer type has a corresponding type specifier that denotes the range of integers to which it corresponds. The names of the type specifiers are **corba:long**, **corba:short**, **corba:ulong**, **corba:ushort**, **corba:longlong**, and **corba:ulonglong**.

3.3.8 Floating point types

The floating point types **float**, **double**, and **long double** map to Lisp types named **corba:float**, **corba:double** and **corba:longdouble** respectively. These types must be subtypes of the type **real**. They must allow representation of all numbers specified by the corresponding CORBA types.

3.3.9 fixed

The fixed point type is mapped to the lisp type named **corba:fixed**. This type must be a subtype of the lisp type **rational**.

3.4 Introduction to named types

We now discuss the mapping of types that are named. We begin with a discussion of terminological issues.

3.4.1 Naming terminology

Notation for naming can be confusing, so some care is needed. Our specification is not formally rigorous, but we have tried to illustrate enough points with examples so that situations likely to arise in practice can be handled.

3.4.1.1 IDL naming terminology

By the *IDL name* of an IDL entity we mean the string that is the simple name of that entity.

An IDL entity can be declared at the top-level or nested inside some other IDL entity. We say that the outer IDL entity *encloses* the inner one.

We will sometimes elide the quotation marks in describing the names of IDL (and other entities) when no confusion is likely to result.

IDL Example

```
module A{  
  interface B{  
    struct c {long foo;};};
```

The name of the **struct** is the string “c”. The name of the **interface** is the string “B”. The name of the **module** is the string “A”. The name of the **struct** member is the string “foo”. The innermost enclosing IDL entity of the **struct** is the **interface** named “B”. The innermost enclosing **module** of the **struct** is the **module** named “A”.

3.4.1.2 Lisp naming terminology

The *name* of a symbol is a string used to identify the symbol.

Packages are collections of symbols. A symbol has a *home package*, which also has a name. A package can be named by a symbol or a string. We sometimes loosely say “the package x” when we mean “the package named by x”. A package may have nicknames and we will consider that the nicknames of a package name the package.

Unless otherwise stated, we will assume that distinct package names refer to distinct packages.

Symbols are notated by prefixing the name of the home package of the symbol to the character ‘:’ to the name of the symbol. Case is not significant when this notation is used.

Thus, all symbols generated by this mapping are external symbols of their home package.

A symbol can name a function, a package, a class, a type, a slot, or a variable. These namespaces are disjoint.

All alphabetic characters in the names of symbols used in this document are upper-case unless otherwise stated.

Thus, the names notated here are implicitly converted to uppercase when they name a symbol.

For example, when we write

the symbol named **hello-goodbye**

or

the symbol **hello-goodbye**

we actually mean the symbol whose name is the string “HELLO-GOODBYE”.

3.5 *Distinguished packages*

This document will refer to to kinds of packages:

- The first kind comprises those packages defined explicitly by this specification.
- The second kind of package comprises those packages created as a result of compiling user IDL code.

The first kind of packages consists of these three distinct packages: the *root package*, the *corba package*, and the *operation package*.

The names of these packages are described below.

The name of the root package is the string “OMG.ORG/ROOT”.

The name of the corba package is “OMG.ORG/CORBA”.

The name of the operation package is the string “OMG.ORG/OPERATION”.

The precise semantics of these three packages is described below. Informally, the root package is the package in which Lisp names corresponding to IDL definitions not contained in a top-level module are interned. The corba package is the package in which Lisp names corresponding to IDL definitions and pseudo-IDL definitions in the CORBA module are interned. The operation package is the package into which names of Lisp functions corresponding to IDL operations are interned.

In addition, this specification makes use of the standard Common Lisp packages named “KEYWORD” and “COMMON-LISP”.

3.5.1 Nicknames for distinguished packages

An implementation is expected to support the addition of nicknames for a package via the standard common lisp nicknames facility. An ORB should support the following default nicknames:

- For the package “OMG.ORG/CORBA” the default nickname shall be “CORBA”.
- For the package “OMG.ORG/OPERATION” the default nickname shall be “OP”.

This document will use these nicknames without comment.

3.6 Scoped names and scoped symbols

Many of the Lisp entities we consider will be named according to the scoped naming convention described in this section. In particular, the following entities will be mapped according to this naming convention:

- **interface**
- **union**
- **enum**
- **struct**
- **exception**
- **const**
- **typedef**

A scoped symbol will be associated with the IDL entity, and it is this scoped symbol that will name the Lisp value generated by the given IDL entity.

3.6.1 Definitions

For any named IDL entity *I* there is a Lisp symbol *S* called the *scoped symbol* of *I*.

The *scoping separator* is the string “/”.

If *I* is a top-level **module**, then the name of *S* is the name of *I*.

If *I* is a **module** nested within another **module** *J*, then the name of *S* is the concatenation of the name of the scoped symbol of *J*, the scoping separator, and the name of *I*.

The home package of the scoped symbol of a **module** is **:keyword**.

Suppose *I* is a named IDL entity that is not a **module**. The name of the scoping symbol *S* of *I* is determined as follows.

If the declaration of *I* is enclosed inside another IDL entity *J* that is not a **module**, then the name of *S* is the concatenation of the name of the scoping symbol for *J*, the scoping separator, and the name of *I*. Otherwise the name of *S* is the name of *I*.

If *I* is enclosed in a **module** *M* then the home package of *S* is named by the scoped symbol for *M*. Otherwise the home package for *S* is the root package.

3.6.1.1 Examples of scoping symbols

First we consider a simple example:

IDL

```
module a {
  interface foo {};
```

The scoped symbol of the module is **:a**. Thus, the home package of this symbol is **:keyword** and the name of the symbol is the string “A”.

The scoped symbol of the interface is the symbol **a:foo**. Thus, the name of the symbol is the string “FOO” and the home package of the symbol is the package whose name is the string “A”.

IDL

```
module a {
  interface outer {
    struct inner {
      in long member;};};
```

Here the scoped symbol for the **module** is **:a**, the scoped symbol for the **interface** is **a:outer**, and the scoped symbol for **struct** is **a:outer/inner**.

IDL

```
module a{
  module b{
    interface c{
      struct d{
        long foo;};};};
```

The scoped symbol for the **struct** is **a/b:c/d**. The scoped symbol for the **struct** member is **a/b:c/d/foo**.

3.7 The package_prefix pragma

A *package_prefix* pragma has the form

```
#pragma package_prefix string
```

where *string* is an IDL string literal. For example

#pragma package_prefix “COM.FRANZ-”

A **package_prefix** pragma affects the mapping of all top-level modules whose definition textually follows that **pragma** in the IDL file: the name of the scoping symbol for such a top-level module is the concatenation of the given **package_prefix** with the name of the module.

All OMG system IDL files, such as the IDL files for CORBA Services and CORBA facilities, are considered to have been defined with an implicit **package_prefix** of “OMG.ORG/”. This name and convention was chosen to be consistent with the way in which system repository ID specifiers are determined. Packages corresponding to modules within the scope of such an implicit **package_prefix** will have default nicknames that are the name of the module without any prefix.

IDL

```
#pragma package_prefix “COM.FRANZ-”
module a{
    module b{
        interface c{};};
```

The scoped symbol for the interface is **COM.FRANZ-A/B:C**.

3.8 Mapping for interface

An IDL **interface** is mapped to a Lisp **class**. The name of this **class** is the scoped symbol for the **interface**.

The direct superclasses of a generated Lisp class are determined as follows. If the given IDL interface has no declared base interfaces, the generated class has the single direct superclass named **corba:object**.

Otherwise, the generated Lisp class has direct superclasses that are the generated classes corresponding to the declared base interfaces of the given interface.

The Lisp value **nil** can be passed wherever an object reference is expected.

An IDL interface is also mapped into server side classes. The server classes are described in the chapter on Server Side mapping.

3.8.1 Example

3.8.1.1 IDL

```
module example{
  interface foo {};
  interface bar {};
  interface fum : foo,bar {};}

```

3.8.1.2 generated Lisp

```
(defclass example:foo(corba:object))
(defclass example:bar(corba:object))
(defclass example:fum (example:foo example:bar))

```

3.9 Mapping for operation

This section discusses only how the user is to invoke mapped operations, not how the user is to implement them. The implementation of operations is discussed in the server chapter.

An IDL operation is mapped to a Lisp function named by the symbol whose print-name is given by the name of the operation interned in the operation package.

We will assume that all operation names have been appropriately imported into the current package in the examples.

Thus, when an example is given in which there is a reference to the symbol naming the mapped function corresponding to an IDL operation, the package of that symbol will be assumed to be the operation package. Common Lisp provides a number of facilities for the implementation of this functionality and for handling name conflicts; we expect in addition the ORBs will provide various convenience functions for this.

3.9.1 Parameter passing modes

The function defined by the IDL operation expects actual arguments corresponding to each formal argument that is declared **in** or **inout**, in the order in which they are declared in the IDL definition of the operation.

3.9.2 Return values

The function defined by the IDL operation returns multiple values. The first (i.e., the zeroth) value returned is that value corresponding to the declared return value, unless the declared return value is **void**. Following the value corresponding to the declared return value, if any, the succeeding returned values correspond to the parameters that were declared **out** and **inout**, in the order in which those parameters were declared in the IDL declaration.

Note that this implies that generated functions corresponding to operations declared **void** which have neither **out** nor **inout** formal parameters return zero values.

3.9.3 *one-way*

Operations declared **oneway** are mapped according to the above rules.

3.9.4 *Efficiency optimization: using macros instead of functions*

A conforming implementation may map an operation to a macro whose name and invocation syntax are consistent with the above mapping. For the sake of terminological simplicity, however, this document will continue to refer to mapped operations as “functions”.

3.9.5 **exception**

An invocation of a function corresponding to a given IDL operation may result in the certain conditions being signalled, including the conditions generated by the exceptions declared in the **raises** clause of the operation, if any. Such conditions are signalled in the dynamic environment of the caller.

An invocation of a function may also result in the signalling of conditions corresponding to system exceptions.

3.9.6 **context**

For each context name declared by an operation, the mapped function accepts a corresponding keyword argument whose name is the name of that context name. If two context names differ only in case, then the corresponding keywords have names identical to the context names, i.e., without case translation. Otherwise, case translation is performed as usual.

3.9.7 Example

3.9.7.1 IDL

```

module example {
  interface face {
    long sample_method (in long arg);
    void voidmethod();
    void voidmethod2(out short arg);
    string method3 (out short arg1,inout string arg2,in boolean arg3);
  };

```

3.9.7.2 generated Lisp

```

(defpackage :example)
(defclass example:face(corba:object))
;...

```

3.9.7.3 usage

; Suppose x is bound to a value of class example:face.

```

(sample_method x 3)
> 24

```

```

(voidmethod x)
> ; No values returned

```

```

(voidmethod2 x)
> 905 ; This is the value corresponding to the out arg

```

```

(method3 x "Argument corresponding to arg2" T)
> "The values returned" -23 "New arg2 value"

```

; The Lisp construct multiple-value-bind can also be used to recover these values.

```

(multiple-value-bind (result arg1 arg2)
  (method3 x "Argument corresponding to arg2" T)
  (list result arg1 arg2))
> ("The values returned" -23 "New arg2 value")

```

3.10 Mapping for attribute

attribute is mapped using a naming convention similar to that for operation.

3.10.1 readonly attribute

An **attribute** that is declared with the **readonly** modifier is mapped to methods whose name is the name of the given **attribute** and whose home package is the operation package.

This method is specialized on the class corresponding to the IDL interface in which the **attribute** is defined.

3.10.2 normal attribute

attributes that are not declared **readonly** are mapped to a pair of methods that follow the convention used for default slot accessors generated by **defclass**.

Specifically, a reader-method is defined whose name follows the convention for **readonly attributes**. A writer is defined whose name is **(self name)** where **name** is the name of the defined reader-method.

3.10.3 Example

3.10.3.1 IDL

```
module example{
  interface attributes {
    attribute string attr1;
    readonly attribute long attr2;};}
```

3.10.3.2 Usage

```
;; Assume x is bound to an object of class example:attributes
(attr2 x)
> 40001
(attr1 x)
> "Sample"
(self (attr1 x) "New value")
(attr1 x)
> "New value"
```

3.11 Mapping of module

An IDL **module** is mapped to a Lisp **package** whose name is the name of the scoped symbol for that **module**.

3.11.1 Example

3.11.1.1 IDL

```
interface outer_interface {};  
  
module example {  
  interface inner_interface {};  
  module nested_inner_example {...  
    interface nested_inner_interface{};  
    module doubly_nested_inner_example{...};  
  };  
}
```

3.11.1.2 generated Lisp

```
(defpackage :example)  
(defpackage :example/nested_inner_example)  
(defpackage :example/nested_inner_example/doubly_nested_inner_example)  
  
(defclass omg.root:outer_interface...)  
(defclass example:inner_interface ...)  
(defclass example/nested_inner_example:nested_inner_interface ...)
```

3.12 Mapping for **enum**

An IDL **enum** is mapped to a Lisp type whose name is the corresponding scoped symbol.

Each member of the **enum** is mapped to a symbol with the same name as that member whose home package is the keyword package.

3.12.1 Example

3.12.1.1 IDL

```
module example{
    enum foo {hello, goodbye, farewell};
};
```

3.12.1.2 generated Lisp

```
(defpackage :example)
(deftype example:foo ()
  '(member :hello :goodbye :farewell))
```

3.12.1.3 usage

```
(typep :goodbye 'example:foo)
> T
(typep :not-a-member 'enumexample:foo)
> nil
```

3.13 Mapping for **struct**

An IDL **struct** is mapped to a Lisp class whose name is the corresponding scoped symbol. Each member of the **struct** is mapped to an initialization keyword, a reader, and a writer.

The initialization keyword is a symbol whose name is the name of the member and whose package is the keyword package.

The reader is named by a symbol that follows the conventions for **attribute** accessors. In the case of a reader its package is the operation package, and its name is the name of the member.

The writer is formed by using **setf** on the generalized place named by the reader.

The type **corba:struct** is defined to be the union of all such generated types.

An IDL struct has a corresponding constructor whose name is the same as the name of mapped Lisp type. This constructor takes keyword arguments whose package is the keyword package and whose name equals the name of the corresponding member.

3.13.1 Example

3.13.1.1 IDL

```
module structmodule{
  struct struct_type {
    long field1;
    string field2;
  };
};
```

3.13.1.2 generated Lisp

```
(defpackage :structmodule)
(defclass structmodule:struct_type (corba:struct)
  ((field1 ...)
   (field2 ...)))
```

3.13.1.3 usage

```
(setq struct (structmodule:struct_type
              :field1 100000
              :field2 "The value of field2"))
```

```
(field1 struct)
> 100000
```

```
(setf (field1 struct) -500)
(field1 struct)
> -500
```

3.14 Mapping for union

An IDL **union** is mapped to a Lisp **class** named by the the corresponding scoped symbol. This class inherits from **corba:union**

The value of the discriminator can be accessed using the accessor function named **union-discriminator** whose home package is the operation package and an initialization argument named **:union-discriminator**.

The value can be accessed using the accessor function named **union-value** in the operation package with initialization argument **:union-value**.

An IDL union has a corresponding constructor whose name is the same as the name of the type. This constructor takes two constructors whose names are **:union-value** and **:union-discriminator**.

3.14.1 Member accessors

Each union member has an associated constructor and accessor.

The symbol-name of the name of the constructor corresponding to a particular member is the concatenation of the name of the union constructor to the scoping separator to the name of the member. The home package of the name of the constructor corresponding to a particular member is the home package of the name of the union constructor.

A constructor corresponding to a member takes a single argument, the value of the union. The discriminator is set to the value of the first case label corresponding to that member.

It is an error if a member reader is invoked on a union whose discriminator value is not legal for that member. The member writer sets the discriminator value to the first case label corresponding to that member.

The default member is treated as if it were a member named default whose case labels include all legal case labels that are not case labels of other members in the union.

3.14.2 Example

3.14.2.1 IDL

```
module example {  
  enum enum_type {first,second,third,fourth,fifth};  
  union union_type switch (enum_type) {  
    case first: long win;  
    case second: short place;  
    case third:  
    case fourth: octet show;  
    default:  boolean other;  
  }; };
```

3.14.2.2 *generated Lisp*

```
(defpackage :example)
(defclass example:union_type (corba:union)
  (...))
```

3.14.2.3 *Usage*

```
(setq union (example:union_type
             :union-discriminator :first
             :union-value -100000))

(union-value union)
> -100000
(union-discriminator union)
> :FIRST
(setq same-union (example:union_type/win -100000))
(union-discriminator same-union)
> :FIRST
(setf (show same-union) 3)
(union-discriminator same-union)
> :THIRD
(show same-union)
> 3
(setf (default same-union) nil)
(union-discriminator same-union)
> :FIFTH
```

3.15 *Mapping for **const***

An IDL **const** is mapped to a Lisp **constant** whose name is the scoped symbol corresponding to that **const** and whose value is the mapped version of the corresponding value.

3.15.1 *Example*

3.15.1.1 *IDL*

```
module example {
    const long constant = -321;
};
```

3.15.1.2 *Generated Lisp*

```
(defpackage :example)
(defconstant example:constant -321)
```

3.16 *Mapping for array*

An IDL **array** is mapped to a Lisp array of the same rank. The element type of the mapped **array** must be a supertype of the Lisp type into which the element type of the IDL array is mapped.

Multidimensional IDL arrays are mapped to multidimensional Lisp arrays of the same dimensions.

3.16.1 *Example*

3.16.1.1 *IDL*

```
module example {
  typedef short array1[2][3];
  interface array_interface{
    array1 op();}}

```

3.16.1.2 *Generated Lisp*

```
(defpackage :example)
(deftype example:array1 () (array (2 3)))
;; mapping for the interface...
(defclass example:array_interface...)
```

3.16.1.3 *usage*

```
(setq a2 (op x)) ; Get an array
(aref a2 0 1)   ; Access an element
> 3 ; Just an example, could be any value that is a short
```

3.17 *Mapping for sequence*

An IDL **sequence** is mapped to a Lisp **sequence**. Bounds checking shall be done on bounded **sequences** when they are marshaled as parameters to IDL **operations**.

An implementation is free to specify the type of the mapped list more specifically.

Suppose **foo** is an IDL data type and let **L** be the corresponding Lisp type.

This means that anywhere a parameter of type **sequence<foo>** is expected, either a **vector** all of whose elements are of type **L** or a **list** all of whose elements are of type **L** may be passed.

Conversely, when such a **sequence** is returned from an operation invocation, this document specifies no type restriction on the returned value other than that it is a **sequence** all of whose elements are of type **L**.

In practice, it is likely that an ORB will marshal and unmarshal **sequence** as appropriately specialized **vector** unless the user provides specific information that this behavior is not desired.

This specification describes a number of functions created by the IDL mapping whose name is a symbol in the IDL package: union member accessors, struct member accessors, attribute accessors, operation mappings, and so on. Whenever such a function is defined, two auxiliary functions are also defined, a *list-coercer* and a *vector-coercer*. The name of the list-coercer is the concatenation of the name of the original function to the string “-LIST”; the name of the vector-coercer is the concatenation of the name of the original function to the string “-VECTOR”; each function name has home package of the operation package.

The effect of invoking the list-coercer corresponding to a particular function on arguments is equivalent to the effect of coercing the result of invoking the original function on the given arguments to the type **list**; similarly for the effect of invoking the vector-coercer on arguments.

3.17.1 Example

3.17.1.1 IDL

```
module example {
  typedef sequence< long > unbounded_data;
  interface seq{
    boolean param_is_valid(in unbounded_data arg);
  };
};
```

3.17.1.2 Generated Lisp

```
(defpackage :example)
(defun unbounded_data_p (sequence)
  (and (typep sequence 'sequence)
        (every #'(lambda (elt)
                    (typep elt 'corba:long))))

(deftype example:unbounded_data()
  '(satisfies unbounded_data-p))

; Let x be an object of type example:seq

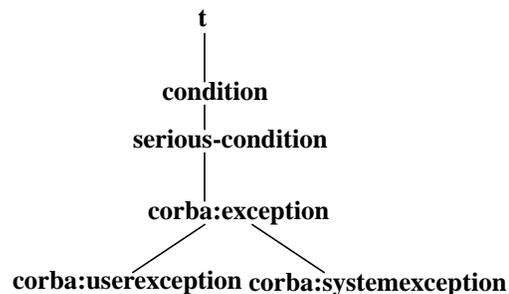
(param_is_valid x '(-2 3))
> T

(param_is_valid x #(-200 33))
> T
```

3.18 Mapping for **exception**

Each IDL exception is mapped to a Lisp condition whose name is the scoped symbol for that exception. User exceptions inherit from a condition named **corba:userexception**. **exception** is a subclass of **serious-condition**.

Figure 3-1 Condition hierarchy for CORBA exceptions



System exceptions inherit from a condition named **corba:systemexception**.

Both **corba:userexception** and **corba:systemexception** inherit from the condition **corba:exception**.

3.18.1 User **exception**

The reader functions and initialization arguments for a condition generated by an IDL **exception** follow the convention for the mapping of IDL **structs**.

3.18.1.1 Example

IDL

```
module example {  
    exception ex1 { string reason; };  
};
```

; generated Lisp

```
(defpackage :example)  
(define-condition example:ex1 (corba:userexception)  
  ((reason :initarg :reason ...))
```

; Usage example

```
(error (example:ex1 :reason "Example of condition"))
```

3.18.2 System **exception**

The standard IDL system **exceptions** are mapped to Lisp **conditions** that are subclasses of **corba:systemexception**. Such generated **conditions** have reader-functions and **initargs** consistent with the IDL definition of these **exceptions**.

3.19 Mapping for **typedef**

IDL **typedef** is mapped to a Lisp type whose name is the scoped symbol corresponding to that **typedef**.

This name of this type denotes the set of Lisp values that correspond to the Lisp type that is generated by the mapping of the IDL type to which the **typedef** corresponds.

However, it is not required to perform recursive checking of the contents of constructed types like **array**, **sequence**, and **struct**.

3.19.1 Example

3.19.1.1 IDL

```
module example{
  typedef unsigned long foo;
  typedef string bar;
```

3.19.1.2 generated Lisp

```
(defpackage :example)
(deftype example:foo () 'corba:unsigned-long)
(deftype example:bar() 'string)
```

3.19.1.3 Usage example

```
(typep -3 'example:foo)
> nil
(typep 6000 'example:bar)
> nil
(typep "hello" 'example:bar)
> T
```

3.20 Mapping for **any**

The IDL type **any** represents an IDL entity with an associated typecode and value. It is mapped to the type **corba:any**, which encompasses all Lisp values with a corresponding typecode.

3.20.1 Constructors

The constructor **corba:any** takes two keyword arguments named **any-value** and **any-typecode**. If **any-typecode** is specified, then **any-value** must be specified. If **any-value** and **any-typecode** are each specified then **any-value** must be a member of the type denoted by **any-typecode**.

An **any** may also be created via the invocation

```
(corba:any :any-typecode val :any-value type).
```

3.20.2 Typecode accessor

The *actual typecode* of a Lisp value **v** is defined as follows.

If **v** was created by an invocation of **corba:any**, then the actual typecode of **v** is the **any-typecode** argument supplied to **corba:any**.

If **v** is a nonnegative integer then the actual typecode of **v** is the the typecode that describes the first Lisp type among (**corba:octet**, **corba:ushort**, **corba:ulong**, **corba:ulonglong**) of which **v** is a member.

Otherwise if **v** is a negative integer then the actual typecode of **v** is that typecode that describes the first Lisp type among (**corba:short**, **corba:long**, **corba:longlong**) of which **v** is a member.

Otherwise if **v** is a member of **corba:float** or **corba:double** then the actual typecode of **v** is **corba:tc_float** or **corba:double** respectively.

Otherwise if **v** is a member of **corba:boolean** then the actual typecode of **v** is **corba:boolean**.

Otherwise if **v** is a **char** then the actual typecode of **v** is **corba:tc_char**.

Otherwise if **v** is a string designator then the actual typecode of **v** is **corba:tc_string**.

Otherwise if **v** is an **array** then then the actual typecode of **v** a typecode describing an array compatible with the contents of **v**.

Otherwise if **v** is a **list** then the the actual typecode of **v** is a typecode describing a **sequence** compatible with the contents of **v**.

Otherwise **v** must be an instance of **corba:object**, **corba:struct** or **corba:union** and the actual typecode is the typecode describing the **exception**, **interface**, **struct**, or **union** of which **v** is an instance. (Such a **v** is said to be *self-typing*).

(**any-typecode v**) is defined to resolve to the actual typecode of **v**.

3.20.3 *value accessor*

If **v** is a number, a string, a sequence, a boolean, or an instance of **corba:enum**, **corba:object**, or **corba:struct** then (**any-value v**) evaluates to a value that is **eql** to **v**.

Otherwise, if **v** is an **any** created via a call to the **corba:any** constructor, then (**any-value v**) resolves to the **any-value** specified in that call.

Otherwise the ORB may signal a **CORBA:BAD_PARAM** exception. This might be necessary, for example, if the ORB received an any containing an instance of a **struct** type for which it does not have enough static information to construct a value of that type. In this case, the value of the **any** can be accessed through the **DynAny** pseudo interface.

3.20.4 *Interaction with GIOP*

For the purpose of GIOP marshalling, a Lisp entity is considered to have the typecode and value corresponding to its actual typecode and actual value.

For example, consider the following IDL:

```
module example{
  interface any_example{
    void foo (in any val);};}
```

Now suppose that **x** is bound to a proxy for a remote implementation of the **example::any_example interface** and suppose requests are forwarded over GIOP to the remote object.

An invocation

```
(foo x 3)
```

will forward to the remote implementation a request to invoke the “foo” method with single parameter an **any** whose typecode is the typecode for octet and whose value is the integer 3.

However, an invocation

```
(foo x (corba:any :any-typecode corba:tc_longlong :any-value 3))
```

will forward to the remote implementation a request to invoke the “foo” method with single parameter an **any** whose **typecode** is the typecode for **long long** and whose value the integer 3.

Thus, the default coercion rules for **any** may be overridden as necessary.

Furthermore, the **DynAny** pseudo interface provides an alternative way to access the values in an **any**.

3.20.5 Additional examples of **any** usage

```
(any-typecode 3)
> <octet typecode>
(any-typecode -1)
> <short typecode>
(any-typecode “foo”)
> <string typecode> ; could also be typecode for an array.
(any-value “foo”)
> “foo”
(any-value nil)
> nil
(any-typecode nil)
> <typecode for boolean>
```

3.21 Mapping Overview

The detailed mapping guidelines for specific types was designed to conform to a small set of uniform principles.

3.21.1 Rule 1: How names of types are formed

If an IDL identifier *I* names a type at the top level of some module named *M*, then the corresponding Lisp type is named *M:I*, that is, the symbol in package *M* whose name is the string “*I*”.

Nested types are separated by the character “/”. Thus, if there is another type *J* defined within the scope of the type named by *I*, the corresponding Lisp symbol is *M:I/J*. This retains consistency with the way in which repository ID’s are formed.

3.21.2 Rule 2: How names of operations are formed

The rule for operation package mapping is simpler: All symbols that correspond to Lisp functions that correspond to IDL operations are interned in a single package. This package can be denoted by “OP”. Thus, *op:foo* denotes the operation named *foo*.

3.21.3 Rule 3: Lisp functions corresponding to IDL types

IDL defines many kinds of types: unions, structs, interfaces, exceptions.

We can think of each of each of these types, informally, as denoting entities with “named slots”. For example, the “named slots” of a struct, union, or exception are its members; the “named slots” of an interface are its attributes.

For each IDL type, there is an associated constructor function that creates a value of that type and there are accessors for each member.

3.21.3.1 The constructor function

The constructor function corresponding to a type is identical to the (fully scoped) name of the type. It takes keyword initialization arguments whose names are the names of the named members of that type; these initialize the given members.

3.21.3.2 Accessing the members

Each “named slot” defines two functions: a reader and a writer. The reader has the same name as the “named slot”. The writer uses the standard (*setf* name) convention familiar to Lisp users. Of course, the home package of the reader is, as for all such function names, the package OP.

3.21.3.3 Notes

In applying Rule 3, it is important to note that not all of the associated functions make sense for all of the types. For example, there is obviously no constructor function defined for an interface, nor are there writer functions defined for attributes declared *readonly*.

I

4.1 Introduction¹

Pseudo-objects are constructs whose definition is usually specified in “IDL”, but whose mapping is language specified. A pseudo-object is not (usually) a regular CORBA object.

For each of the standard IDL pseudo-objects we either specify a specific Lisp language construct or we specify it as a **pseudo interface**.

We have chosen the option allowed in the IDL specification section 4.1.3 to define **Status** as **void** and have eliminated it for the convenience of Lisp programmers.

A Pseudo-object differs from a regular CORBA object in the following ways:

- It is not represented in the Interface Repository.
- It may not be passed as a parameter to an operation expecting a CORBA Object.
- It may not be returned as a CORBA Object.
- It may not be stored in an **any**.
- It may not be safely subclassed by user code, if it is represented as a class.

4.2 Certain exceptions

The standard CORBA PIDL uses several **exceptions**, **Bounds**, **BadKind**, and **InvalidName**.

(define-condition corba:bounds (corba:userexception)...)

1.This chapter has not been fully updated from the 2.1 pseudo IDL to the 2.2 pseudo IDL.

```
(define-condition corba:typecode/badkind(corba:userexception)...)
(define-condition corba:typecode/bounds(corba:userexception)...)
(define-condition boa:invalidname (corba:useexception)...)

```

4.3 Environment

The **Environment** is used in request operations to make **exception** information available.

Since **conditions** in Lisp are first class objects, we see no reason not to define **Environment** simply as an **exception**:

```
(deftype corba:environment() 'corba:exception)
```

4.4 NamedValue

A **NamedValue** describes a name, value pair. It is used in the DII to describe arguments and return values, and in the **context** routines to pass property, value pairs.

We map this as if it were a normal **struct** as specified by the IDL using the IDL in module CORBA:

```
typedef unsigned long Flags;
typedef string Identifier;
const Flags ARG_IN = 1;
const Flags ARG_OUT = 2;
const Flags ARG_INOUT = 3;
const FLAGS CTX_RESTRICT_SCOPE = 15;
```

```
struct NamedValue{
    Identifier name;
    any argument;
    long len;
    Flags arg_modes;}
```

4.5 NVList

A **NVList** is used in the DII to describe arguments and in the context routines to describe context values. An **NVList** is mapped to an object of class **CORBA:NVList** whose pseudo-IDL is given below.

```
pseudo interface NVList {
  readonly attribute unsigned long count;
  NamedValue add (in Flags flags);
  NamedValue add_item (in Identifier item_name, in Flags flags);
  NamedValue add_value (in Identifier item_name,
                        in any val,
                        in Flags flags);
  NamedValue item (in unsigned long index) raises (CORBA::Bounds);
  void remove (in unsigned long index) raises (CORBA::Bounds);
}
```

4.6 Context

A **Context** is used in the DII to specify a **context** in which **context** strings must be resolved before being sent along with the request invocation.

It is mapped to a class **corba:context** whose operations are as specified in the PIDL for this class.

```
pseudo interface Context {
  readonly attribute Identifier context_name;
  readonly attribute Context parent;
  Context create_child (in Identifier child_ctx_name);
  void set_one_value (in Identifier proptype, in any propvalue);
  void set_values (in NVList values);
  void delete_values (in Identifier proptype);
  NVList get_values (in Identifier start_scope,
                    in Flags op_flags,
                    in Identifier pattern);
}
```

4.7 Request

A **Request** is mapped to an instance of class **CORBA:request** according to the IDL:

```
typedef sequence<Exception> ExceptionList;
typedef sequence<Context> ContextList;
pseudo interface Request {
    readonly attribute Object target;
    readonly attribute Identifier operation;
    readonly attribute NVList arguments;
    readonly attribute NamedValue result;
    readonly attribute Environment env;
    readonly attribute ExceptionList exceptions;
    readonly attribute ContextList contexts;
    attribute Context ctx;
    any add_in_arg();
    any add_named_in_arg (in string name);
    any add_inout_arg();
    any add_named_inout_arg(in string name);
    any add_named_out_arg(in string name);
    void set_return_type(in TypeCode tc);
    any return_value();

    void invoke();
    void send_oneway();
    void send_deferred();
    void get_response();
    boolean poll_response();
}
```

4.8 ServerRequest

ServerRequest is used in the DSI. It is to be mapped according to the IDL to the Lisp class named **CORBA:ServerRequest**.

```
pseudo interface ServerRequest{
    Identifier op_name();
    Context ctx();
    void params (in NVList parms);
    void result (in any res);
    void except (in any ex);
}
```

4.9 TypeCode

The deprecated **parameter** and **param_count** methods are not mapped.

A **TypeCode** is an instance of the class named **CORBA:TypeCode**. It follows the pseudo IDL below.

```

enum TCKind{
tk_null, tk_void, tk_short, tk_long, tk_ushort, tk_ulong, tk_float, tk_double,
tk_boolean, tk_char, tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
tk_struct, tk_union, tk_enum, tk_string, tk_sequence, tk_array, tk_alias,
tk_except, tk_longlong, tk_ulonglong, tk_longdouble, tk_wchar, tk_wstring,
tk_fixed};

pseudo interface TypeCode {
    exception Bounds{};
    exception BadKind{};
    boolean equal (in TypeCode tc);

//for objref, struct, union, enum, alias, and except
TCKind kind();
RepositoryId id() raises (BadKind);
Identifier name() raises (BadKind);

//for struct, union, enum, and except
unsigned long member_count() raises (BadKind);
Identifier member_name(in unsigned long index) raises (BadKind, Bounds);

//for struct, union, and except
TypeCode member_type (in unsigned long index) raises (BadKind, Bounds);

//for union
any member_label(in unsigned long index) raises (BadKind, Bounds);
TypeCode discriminator_type() raises (BadKind);
long default_index() raises (BadKind);

//for string, sequence, and array
unsigned long length() raises (BadKind);
TypeCode content_type() raises (BadKind);

```

4.10 ORB

4.10.1 ORB initialization

An ORB is initialized via the ORB_init pseudooperation in the CORBA module:

This pseudooperation simply takes as argument various implementation-defined keywords.

4.10.2 ORB pseudo-object

The **ORB** is mapped according to its pseudo-IDL definition. This includes the following IDL:

```
pseudo interface ORB {
    exception InvalidName{};
    typedef string ObjectId;
    typedef sequence<ObjectId> ObjectIdList;
    ObjectIdList list_initial_services();
    Object resolve_initial_references(in ObjectId object_name)
    raises(InvalidName);

    string object_to_string (in Object obj);
    Object string_to_object (in string str);

    NVList create_list(in long count);
    NVList create_operation_list(in OperationDef oper);
    NamedValue create_named_value(in String name, in any value, in Flags
    flags);
    Context get_default_context();
    void send_multiple_requests_oneway(in RequestSeq req);
    void send_multiple_requests_deferred(in RequestSeq req);
    boolean poll_next_response();
    Request get_next_response();

    //typecode creation

    TypeCode create_struct_tc (
        in RepositoryId id,
        in Identifier name,
        in StructMemberSeq members);

    TypeCode create_union_tc(
        in RepositoryId id,
        in Identifier name,
        in Typecode discriminator_type,
        in UnionMemberSeq members);

    Typecode create_enum_tc(
        in RepositoryId id,
        in Identifier name,
        in EnumMemberSeq members);

    TypeCode create_alias_tc(
        in RepositoryId id,
        in Identifier name,
        in TypeCode original_type);

    TypeCode create_exception_tc(
        in RepositoryId id,
        in Identifier name,
        in StructMemberSeq members);

    TypeCode create_interface_tc(
        in RepositoryId id,
```

in Identifier name);

```
TypeCode create_string_tc (in unsigned long bound);
TypeCode create_wstring_tc (in unsigned long bound);
TypeCode create_recursive_sequence_tc (
    in unsigned long bound,
    in unsigned long offset);
TypeCode create_array_tc (
    in unsigned long length,
    in TypeCode element_type)
```

4.11 Object

The IDL **Object** type is mapped to the class **corba:object**. It supports the operations defined in the pseudo-IDL for this type.

The **is_nil** pseudo operation is mapped to a function named **op:is_nil** which may be applied to the lisp value **nil**.

The **duplicate** and **release** pseudo-operations are unnecessary in the Lisp mapping and are not mapped.

```
pseudo interface Object{
    void create_request(
        in Context ctx,
        in Identifier operation,
        in NVList arg_list,
        inout NamedValue result,
        out Request request,
        in Flags req_flags);
    InterfaceDef get_interface();
    boolean is_nil();
    boolean is_a (in string logical_type_id);
    boolean non_existent();
    boolean is_equivalent (in Object other_object);
    unsigned long hash (in unsigned long maximum);
```

4.12 Principal

The **Principal** interface is deprecated and is not mapped.

4.13 DynAny

The **DynAny** pseudo interface is mapped according to its pseudo IDL without any modification. A **DynAny** is an instance of the class **corba:DynAny**.

4.14 The IDL Compiler

The IDL compiler uses the following top-level pseudo-IDL definition in the **CORBA** module:

```
typedef string pathname_designator;  
Repository idl (pathname_designator path);
```

The Lisp mapping is to the function named **corba:idl** that takes a single argument, a pathname designator for an IDL source file.

The effect of invoking **corba:idl** on a pathname designator is to define within the Lisp world all data types, packages, proxies, and stubs defined by the denoted IDL file. This may entail redefining classes or types.

If the Lisp mapping requires that package named **P** be created, and there is already a package **Q** with **P** as one of its names or nicknames in the current Lisp world, then the package **Q** is used everywhere the package named **P** is required. Previously existing symbols interned in **Q**, or other attributes of **Q** such as the packages it uses, are not affected. However, if a symbol is interned in, but not exported by, **Q**, and if the mapping requires this symbol be external, its visibility is appropriately modified as a result of the **corba:idl** mapping.

The object returned is an object of type **corba:repository** and represents an Interface Repository representing the IDL file given as input. The precise semantics of this representation is implementation dependent, although it should contain objects at least that represent each definition in the input IDL file, unless it returns **nil**.

Implementations may freely add additional keywords to **corba:idl** to express additional functionality. For example, the implementation may augment this specification with keywords to describe the names of packages into which IDL entities are mapped, the visibility of symbols, and preprocessor directives.

5.1 Introduction

This chapter discusses how implementations create and register objects with the ORB runtime.

5.2 Mapping of native types

Specifically, the native type `PortableServer::Servant` is mapped to the Lisp class named `PortableServer:Servant`. The native type `PortableServer::ServantLocator::Cookie` is mapped to the Lisp type `PortableServer:ServantLocator/Cookie`.

5.3 Implementation objects

An implementation of an IDL interface `I` corresponding to a Lisp class named `I` should inherit, directly or indirectly, from the classes named `I` and `PortableServer:Servant`.

5.4 Servant classes

An interface corresponding to a class named by a Lisp symbol `s` with package `p` and name `n` may be implemented by extending the class named by the symbol whose package is `p` and whose name is the concatenation of `n` to the string “**-SERVANT**”.

For each **attribute** in the **interface**, the associated servant class has a slot whose name is the name of the attribute and whose home package is the operation package.

If the interface has no base interfaces, then the associated skeleton class has as direct superclasses the class corresponding to the given interface and the class named **corba:servant**.

Otherwise, if the interface has base interfaces named **A**, **B**, **C**... then its associated servant class has as direct superclasses the class corresponding to the given interface and the servant classes corresponding to **A**, **B**, **C**...

5.4.1 Note on proxies

An ORB that supports proxies is encouraged to use a similar inheritance hierarchy for proxies, with “**servant**” replaced by “**proxy**” in all generated classes above. This is intended to help allow more portable auxiliary method definition.

5.5 Defining methods

The only portable way to implement an operation on a servant class is by use of the **corba:define-method** macro.

The syntax of **corba:define-method** is intended to follow as closely as possibly the syntax of the Lisp **defmethod** macro.

5.5.1 Syntax of **corba:define-method**

corba:define-method function-name {method-qualifier}* corba-specialized-lambda-list form*

function-name::= {operation-name | (self operation-name)}
operation-name:: symbol
method-qualifier::={:before | :after | :around}
corba-specialized-lambda-list ::= self-lambda-list | normal-lambda-list
self-lambda-list ::= (argument-specifier receiver-specifier)
normal-lambda-list ::= (receiver-specifier {parameter-specifier}* context-list)
context-list ::= {} | {&key {context-identifier}+}
context-identifier ::= symbol
receiver-specifier ::= (receiver-name receiver-class)
receiver-name ::= symbol
receiver-class ::= symbol
parameter-specifier ::= symbol

5.5.2 Description

This **corba:define-method** macro is used to implement an operation on an interface.

operation-name is a symbol whose name is the name either of an operation or of an attribute declared in an IDL interface implemented by the class named by the symbol **receiver-class**.

The number of **parameter-specifiers** listed in the **normal-lambda-list** must equal the combined number of **in** and **inout** parameters declared in the signature of the operation denoted by the **function-name**, or **0** if the operation is an attribute. If the **function-name** is a list whose **car** is **self**, the corresponding **operation-name** should name an attribute that is not **readonly**.

If **function-name** denotes an operation, then the effect of **corba:define-method** is to inform the ORB that requests for the operation on instances of the class **receiver-class** should return the value or values returned by the body forms of the **define-method** macro, executed in a new lexical environment in which each **parameter-specifier** is bound to the actual parameters and in which each **context-identifier** is bound to the value of the corresponding **context** variable.

The operation of **corba:define-method** in the case in which **function-name** names an attribute is analogous.

The behavior of auxiliary specifiers and of dispatch is the same as their corresponding action under **defmethod**.

Note that the syntax of **corba:define-method** is a strict subset of that of **defmethod**: every legal **corba:define-method** invocation is also a legal **defmethod** invocation. The main difference between them is that **corba:define-method** only allows specialization on the first argument.

An implementation is free to extend the syntax of **corba:define-method**, for example to allow type-checking, interlocking, or multiple dispatch.

5.6 Examples

5.6.1 Example: A Named Grid

The first example shows how one might encapsulate a “named-grid”, which is a grid of strings.

5.6.1.1 IDL

This is the IDL of the interface to a named grid of strings.

```
module example{
  interface named_grid{
    readonly attribute string name;
    string get_value ( in unsigned short row,
                      in unsigned short column);
    void set_value ( in unsigned short row,
                    in unsigned short column,
                    in string value);
  }
}
```

5.6.1.2 *Generated Lisp code*

The IDL compiler might generate a class corresponding to the **example::named_grid** interface using code something like this:

```
(defpackage :example)
(defclass example:named_grid(corba:object())
```

5.6.1.3 *Servant class*

In order to implement the IDL interface, the user would extend the class **example:named_grid-servant**.

```

;;Sample implementation of named_grid
(defclass grid-implementation (example:named_grid-servant)
  (
    (grid :initarg :grid
          :initform (make-array '(2 3) :initial-element "Init")))

```

5.6.1.4 Implementation of the IDL operations

The **corba:define-method** macro is used to define the methods that implement each of the operations defined in the IDL interface. Note that the reader method and **initarg** corresponding to the **attribute** name was already defined by the **servant**.

These implementations do not perform any argument or range checking, which a production system would, of course, perform.

The implementation is free to define other methods on the class, including **print-object** methods and **auxiliary** methods for **initialize-instance**.

```

(corba:define-method get_value ((the-grid grid-implementation)
                               row
                               column)
  (aref (slot-value the-grid 'grid) row column))

(corba:define-method set_value ( (the-grid grid-implementation)
                                row
                                column
                                value)
  (setf (aref the-grid row column) value))

```

5.6.1.5 Usage example

Once the implementation class is defined, it can be instantiated and its instances treated as a normal CLOS object. In particular, such instances can be passed to remote ORB servers which expect an object implementing the IDL **named_grid** interface. The invocation of the methods corresponding to IDL **operations** does not depend on whether the object is an instance of the servant class or is simply a proxy for another object (perhaps implemented in another language).

This usage example does not discuss registration of the object with the ORB.

; create a named grid

**(setq grid (make-instance 'example:grid-implementation :name "Example of a
grid"))**

(name grid)

> "Example of a grid"

(set_value grid 0 1 "Hello")

> ; No values returned

(get_value grid 0 1)

> "Hello"

The purpose of this chapter is to explain and to justify the reasoning behind the design choices made. For each key design decision, we discuss alternative proposed decisions or alternative considered design decisions.

This chapter is not normative and is not intended to be included in the finally approved mapping document.

6.1 Introduction

The overall goal of our mapping design was to make a successful Lisp mapping. We wanted the mapping to be widely used in Lisp applications and to be supported by multiple vendors.

We began by studying the existing mappings and in particular determining which mappings appeared to be successful and which did not, and why. We also tried to identify characteristics of Lisp that make it well-suited or ill-suited to use in a CORBA environment. We tried to make sure that our mapping could exploit the traits of Lisp that were well-suited to a CORBA environment while minimizing the traits that were not well-suited to a CORBA environment.

6.1.1 Goals

Within the constraint of faithfully representing IDL semantics, we attempted to satisfy a number of design goals.

6.1.1.1 Ease-of-use

CORBA systems are often cross-platform, cross-language, and cross-vendor. Their development presents certain unavoidable difficulties for the programmer. Our aim was to make the Lisp ORB as simple to use as possible. We strove for a system in which common idioms could be expressed concisely and in which common defaults were

chosen. For example, the skeleton classes automatically generates slots for attributes, operation invocation syntax can be very concise. The any mapping chooses reasonable defaults for most cases, although means to override the defaults are given.

6.1.1.2 *Consistent*

A crucial design goal was that our mapping be as easy to learn to use as possible even for users not expert in Lisp or in CORBA. To achieve this, we aimed for a mapping as consistent as possible.

Attributes or attribute-like values are always mapped the same: to keyword initializers and to accessors with the same name as the attribute. This holds whether or not the attribute-like value corresponds to a true IDL attribute, to a struct member, to a union member, or to an exception member. Constructed types always have a constructor of the same name.

6.1.1.3 *Flexibility*

The mapping should facilitate the production of flexible and dynamically modifiable code. CLOS auxiliary methods and smart proxies are supported; run-time code modification based on dynamically computed repositories is supported.

6.1.1.4 *Performance*

The features described here should not impose undue performance overhead.

6.1.1.5 *Adherence to IDL*

We adhere to IDL conventions as much as possible, even when specifying pseudo-interfaces.

6.1.2 *Lisp Version*

The term “Lisp” is often used colloquially to refer to “Lisp-like” languages—interpreted, high-level, dynamic languages with customizable syntax—such as Scheme or various variants of Common Lisp.

By “Lisp” we will mean exclusively “Common Lisp” as specified by the in X3J13 Committee, *ANSI X3.226-1994, American National Standard for Information Technology—Programming Language—Common Lisp*, ANSI (New York) 1994.

In particular, we do not rely on features that were not present in the ANSI standard. We discuss here three such features that are commonly encountered.

6.1.2.1 *Meta-object protocol*

Although production Lisp systems support the meta-object protocol as described in Art of the Meta-object Protocol (MOP), this protocol was not standardized by X3J13. Therefore, we have been careful to insure that implementation or use of the mapping here in no way relies on portions of the MOP not formally approved by X3J13.

6.1.2.2 *Multithreading*

The X3J13 group did not standardize multiprocessing or concurrency semantics for Lisp, although again all production Lisp systems do include such features. We have therefore not specified a specific multi-processing interface.

6.1.2.3 *Case-sensitivity*

Some vendors of Lisp support “case-sensitive” Lisps of various flavors. This document does not address this issue.

6.1.3 *Reverse mapping*

We have not considered explicitly reverse-mapping issues.

6.1.4 *Compiler interface*

The interface to the compiler is part of the specification so that portable programs can be written that invoke the compilation process.

6.1.5 *Type checking*

We did not specify in most cases that a particular exception be raised if incorrectly typed values are passed to an array, mostly for reasons of simplicity and efficiency of implementation. Of course, an implementation is encouraged to provide as much type-checking as feasible.

6.2 *Overall Design Philosophy*

In evaluating the suitability of committing the standard to a particular design decision, we considered the following:

- Is the decision natural in light of IDL notation and semantics?
- Is the decision natural in view of other languages’ mappings? How did other languages’ mapping treat this problem?
- Does the decision allow reasonable performance?
- Is our decision easy to learn and to remember? Is it consistent with other design decisions in the Lisp mapping?

6.2.1 *Relationship to other mappings*

We relied particularly on three mappings: the C++ mapping, the Java mapping, and the Smalltalk mapping.

The C++ mapping is important because it is one of the most comprehensive mappings and because IDL is very close to C++ in semantic model.

The Java mapping was one of the most recent, is very popular, and is easy to use.

The Smalltalk mapping is one of the most straightforward and consistent of the mappings.

However, we are aware of and studied the C mapping and the Ada mapping. We felt the C mapping was useful primarily as an example of what to avoid: although general and efficient, it is so complicated to use that few ORBs support it.

For example, our any mapping follows the Smalltalk mapping philosophy. Our DII mapping, however, follows the Java mapping.

On the other hand, our handling of names follows the IDL convention for repository ID's rather than the Java convention (again, the C convention was rejected as incomplete).

6.3 *Names*

There are several differences between the IDL and the Lisp namespaces.

6.3.1 *Capitalization*

IDL identifiers are case-sensitive, but two identifiers differing only in case are not allowed to occupy the same namespace.

Although Lisp symbols are also case-sensitive, in practise it is often inconvenient to notate in a Lisp program symbols whose names contain lower-case alphabetic characters, since the Lisp reader by default converts lower-case characters to upper-case characters in symbol names.

Therefore, we have chosen to convert implicitly all IDL identifiers to upper-case.

However, we follow the customary usage of X3J13 in notating symbols using mixed-case—typically lower-case—characters.

6.3.2 *Nesting*

The IDL namespace is deeply nested, although there is only a single “root” namespace.

There are many disjoint Lisp namespaces, each of which is essentially bilevel. We chose to partition the IDL namespaces into a module portion and a non-module portion.

6.3.3 Character set

Lisp symbols typically have names comprising 8-bit characters. However, certain characters, such as the space character, are difficult to work with in practice since they must be escaped for the default Lisp reader.

The situation for IDL identifier is not as clear for the following reasons:

6.3.3.1 International characters

The CORBA 2.1 specification, as has previous CORBA specs, explicitly allows a number of ISO-Latin characters that are not standard ASCII alphabetic characters, such as ß, Æ, and È.

However, no other mapping of which we are aware has provision for mapping symbols containing such characters. In order to remain compatible with existing ORBs, we chose to allow only standard alphanumeric characters and the underscore character in IDL identifiers.

6.3.3.2 Other special characters

Lisp allows punctuation characters such as “/”, “-” and “.” to be part of the character name, while IDL does not. We exploit this fact in a number of instances to avoid the possibility of name clashes.

6.3.3.3 Keywords

Lisp does not have reserved words in the usual sense (although the bindings of certain symbols may not be changed). Therefore, we did not require rules for avoiding clashes with reserved keywords. On the other hand, we did not consider here the issue of generated Lisp package names conflicting with user or system package names. We expect that options may be provided to the compiler to avoid this problem.

6.3.4 Alternative mappings

It would have been possible to choose a name mapping that produced names more familiar to Lisp users. For example, hyphens could have been inserted at case transitions, or underscores could have been converted to hyphens.

6.3.5 Prefixes

We provided a **package_prefix** pragma in order to avoid clashes between IDL module names and generated Lisp package names.

6.4 Mapping of basic types

The mapping for most of the basic types is fairly straightforward, although character-set issues are discussed above.

There are questions in certain cases, however:

6.4.1 **boolean**

We considered mapping this type to the Lisp values defined by **generalized boolean**, which is easier to use in certain cases, but mapping to **boolean** was simpler.

6.4.2 **float and double**

In practice Lisp vendors use IEEE format to represent floating point numbers, but because this representation is not required by the ANSI standard, we chose our mapping to be independent of this.

6.5 Mapping for **struct**

An alternative mapping would map an IDL **struct** directly into a **structure-object**, an object created by the macro **defstruct**. Another reasonable mapping would have been to map a **struct** into a class whose slot accessors obeyed the naming rules for **defstruct** accessors. We chose to maintain consistency with the naming convention for **attribute** in our mapping.

Furthermore, this makes the format of the accessors for **exception** more uniform, as it can simply follow the **struct** format.

However, we have chosen our mapping so that a **structure-class** implementation would not be precluded; we do not insist that **corba:struct** be a subclass of **standard-class**, since for some compilers it could be the case that implementing a **corba:struct** as a **structure-object** would allow a performance improvement.

6.6 Mapping for **exception**

Clearly IDL **exception** should be mapped to Common Lisp **condition**.

Nevertheless, there were several design issues that arose

6.6.1 **condition hierarchy**

It seemed clear that the Lisp **condition** corresponding to CORBA user **exception** and CORBA system **exception** would derive from a common base class named **corba:exception**.

However, it is not clear from which of the standard Lisp **condition** classes the **corba:exception** class would most appropriately derive directly

We considered these options as candidates for the direct superclass of **corba:exception**:

- **condition**, the base class for the Lisp **condition** system
- **error**, the base class for errors.
- **serious-condition**.

We quickly rejected **simple-error**, **simple-condition**, and **warning** as candidates.

The most familiar condition to signal for Lisp programmers would probably be **error**, but the specification does not support this usage.

In particular, the ANSI spec [p. 9-11] states that “The type **error** consists of all conditions that represent errors” where an “error” as used in the last word refers to “a situation in which the semantics of a program are not specified, and in which the consequences are undefined.” We felt that this was too strong a usage for the certain cases of exceptions that are raised.

On the other hand, a **serious-condition** is one which is “serious enough to require interactive intervention if not handled [X3J13 p. 9-10].” This seems like a more appropriate match, and it is the one we chose.

It would certainly be a reasonable mapping for **corba:exception** to inherit directly from **condition**. However, we think that exceptions should be signalled using the Lisp **error** function and not the **signal** function.

The question of the direct superclass of **corba:exception** affects the behavior of condition handlers in whose scope such a condition is signalled, hence the importance of specifying carefully this class.

6.6.2 Naming **exception** classes

We chose to name the classes corresponding to system and user **exceptions** **corba:systemexception** and **corba:userexception** respectively. This naming convention is consistent with the mapping of Java and of C++.

However, the IDL for the **enum** types corresponding to **exception** used in the IDL for the **GIOP** uses an underscore to separate the words: **corba_exception** and **user_exception**, and so **corba:system_exception** and **corba:user_exception** would be an appropriate alternative mappings.

6.6.2.1 Member accessors

We chose a convention for member accessors consistent with that for **struct**, except that of course there were no writer functions defined since conditions are immutable. However, if the **struct** mapping were to change we would reconsider this mapping.

6.7 Mapping for **enum**

A Lisp symbol in the **:keyword** package usually fill the role of **enum** in C-like languages. This mapping has the disadvantage, however, that such values are not self-typing in the sense that they do not encode the name of the enum of which they are a member.

We could have chosen a self-typing mapping as well—languages like Java have two mappings for **enum**, for example—but we chose not to do so.

6.8 Mapping for **union**

An alternative mapping would map the **union** to a base class and each of the branches to concrete subclasses.

We eventually decided to follow closely the Java union mapping, again to shorten the learning curve.

A simpler alternative would have been to map a **union** to a **cons** whose **car** holds the discriminator and whose **cdr** holds the value. However, we felt it was reasonable to maintain the same naming convention for all corba accessor functions.

6.9 Mapping of **module**

The IDL **module** is a name-scoping mechanism in IDL whose corresponding Lisp equivalent is the package. Some separators need to be used between namespace identifiers, since the Lisp package system is not nested.

We chose not to rely on automatic importing of symbols in a **package** corresponding to an outer **module** into the **package** corresponding to the inner **module**, as we felt the potential for confusion outweighed the gain in concision.

Because we are using “/” as a separator for names of components of a nested namespace, we felt the name of the mapping of top-level types should begin with “/”. By default, therefore, top-level modules are in fact mapped to the package whose name is /, although `idl` or `/idl` are reasonable alternatives.

The “/” separator was chosen instead of the “.” separator because that is the separator used by IDL as a scoping separator in repository IDs. However, the “.” is more familiar in this context, since it is used as a scoping separator in the Java mapping, and we are considering modifying the mapping to use “.” as the scoping separator character.

6.10 Mapping for **array**

An IDL **array** is mapped to a Lisp **array**. It would be reasonable to specify formally the declared **:element-type** of the mapped Lisp array, but for simplicity we chose not to in this document.

There is a potential ambiguity in dealing with nested arrays. Consider the following IDL definitions

```
// IDL
typedef short a [2];
typedef a b[3];
typedef short c[2][3];
```

In the mapping, **c** would be mapped to a 2-dimensional **array**, but **b** would be mapped to a one-dimensional **array** of **arrays**. These data structures are disjoint in Lisp and are not accessed using the same syntax.

The problem is that the definition of `ArrayDef` in the interface repository only allows one-dimensional arrays (although the element type can be array). Thus, it might be necessary to map **b** into a Lisp two-dimensional **array** of **integers** as well, so as to interoperate unambiguously with other interface repositories.

Because there are known problems with the treatment of interface repositories in CORBA, we chose not to consider the impact of this problem at this time.

6.11 Mapping for **sequence**

This draft chooses to map IDL **sequence** to the Lisp type **sequence**.

There are several possible alternative mappings.

6.11.0.1 **sequence-to-list**

The simplest mapping to use and to explain is probably the mapping that maps **sequence** to **list**. Unfortunately, such a mapping has substantial performance overhead for cases where the element types are small, such as in the ubiquitous **sequence<octet>**. More important, the **list** data type simply fails to capture gracefully the intended use of **sequence** in certain applications.

6.11.0.2 **sequence-to-vector**

Another natural mapping is for **sequence** to go to **vector**. Although this is an appropriate mapping in cases where the **sequence** elements are small and the sequence size does not change often, it is less appropriate to use when the **sequence** is intended to be modified in size or constructed dynamically.

Of course it would be possible in such cases to map **sequence** to adjustable **array** with fill pointers. These are a subtype of **array** which permitting run-time size modification. Although such arrays are useful in certain applications, they are nevertheless less flexible and are more difficult to use than the **list** datatype for many purposes.

6.11.0.3 *Hybrids*

Some proposed mappings have generally mapped **sequence** to **list**, but have mapped to **array** in certain special cases, e.g. when the elements are small.

6.11.1 *Advantages of our proposal*

- Our proposal is the simplest to use of all the proposals in the common case where the user is writing a client that passes a parameter for which the corresponding parameter was declared as a **sequence**. Indeed, the client can simply use **lists** or **arrays** in the application code, whichever is more convenient.
- Our proposal is more efficient than the **sequence-to-list** in cases where the element types are small or where **vector** is the better data type.
- Our proposal is simpler and more flexible than the hybrid proposal, since there is not artificial demarcation that the user must remember between the mapping conventions.

6.11.2 *Disadvantages of our proposal*

- Our proposal is more difficult to use than the other possibilities in the case where a **sequence** is a return parameter of an **operation**, since the client does not know the type of the **sequence**.
- Our proposal is somewhat more complicated to explain than either the **sequence-to-vector** or the **sequence-to-list** proposal, since **sequence** is not used as often as **list** or **vector** alone.
- Our proposal is slightly more complicated for the implementor of a method, since the method body must be prepared to expect an arbitrary **sequence** (or a syntax in the method definition must allow this conversion to be done automatically).
- Our proposal can lead to problems in verifying the correctness of code that does not correctly handle sequences passed to it; code might fail to work only on certain types of sequences.
- Our proposal imposes a small run-time overhead associated with type-checking of the passed value.

6.11.3 *Conclusion*

It is certainly tempting to fix the mapping of sequence either to vector or to list. However, we believe that the availability of both vector and list data-types in Lisp is quite useful; fixing on either one would constrain functions for which the other would be better suited.

6.12 *Mapping for any*

In the case of **any**, there are several issues to consider: convenience, generality, and accessors.

The any mapping was chosen so that Lisp values can be passed back and forth from operations expecting an any without undue manual coercions, particularly in the common cases where a primitive type is passed.

The special handling of string designators was chosen to avoid ambiguity in passing enum values.

The coercions were chosen so that the typecode would denote the “smallest” containing type in some sense. However, for the sake of implementation simplicity, a list can be passed as **sequence<any>** rather than **sequence<type>** where type is some smaller superset of the types of the contents of the list.

This semantics was chosen particularly to facilitate passing nested lists of primitives.

6.13 Mapping for **typedef**

It seems clear that a **typedef** should map to a Lisp type that contains at least all the values that could be in the range of the mapping of the original IDL type aliased by the given **typedef**. However, whether these sets should coincide—whether a value not in the range mapping should not be in the appropriate type—is problematic for constructed types: how far should the type specifier peer into the object?

These cases arise particularly in handling the mapping for **array**, **sequence**, **struct**, and **union**. It is particularly problematic in the latter two cases since the type specifier is defined automatically from the name of the class defining the **struct** or **union**.

In order to simplify the exposition, we do not mandate special type-checking beyond checking.

6.14 Mapping for **interface**

We chose to map IDL **interface** into **class**.

One alternative would have been to define our own IDL-like object system, for example using a system like flavors. Although the resulting code would have been closer to the object model of IDL, we rejected it as being insufficiently Lisp-like.

Another alternative would have been to require that the metaclass of such mapped classes be a particular metaclass, **corba:metaclass**. Although this approach would yield an elegant and flexible mapping, we rejected it because we did not want to rely on nontrivial features of the MOP in this mapping.

We required implementation classes to inherit from a class named **corba:servant**. It might be more consistent with the POA to name this class **portableservant:servant**, however.

6.15 Mapping for **operation**: *the name*

The proposed **operation** mapping is the most unusual of our proposals, and it diverges from previous mapping proposals. Therefore, it is worth exploring in some detail our motivation.

First, we decided that since **interface** was mapped to **class** it would be natural to map **operation** to method.

We rejected an invocation syntax that relied on macro expansion as these tend either to fail or to be extremely cumbersome when used with **apply**, **funcall**, and related higher-order functions.

The operation package name was a crucial choice. A short name would run the risk of conflicts with user packages; a longer name would be cumbersome.

We chose to support both: the ORB supports a long name, “OMG.OPERATION” by default, but it must support, not only the standard nickname, but even a sort of “super-nickname” of keyword.

We rejected an invocation syntax that relied on reader-macros as being baroque. However, an implementation may, of course, define its own reader macros to ameliorate some of the syntactic awkwardness associated with parts of our mapping.

6.15.1 *Explicit operation mapping*

The most obvious mapping was in fact the mapping chosen by previous Lisp mappings: Simply map the **operation** to a function or method whose name is the fully scoped name of the **operation**, and which takes as first argument the receiver, as in the following example. (Actually the most obvious mapping simply disregards name-clash and package issues entirely).

```
//IDL
module example{
  interface foo{
    long op1(in short arg);}
}
```

generated Lisp

```
(defmethod example:foo/op1 ((this example:foo) arg)
  ... ; implementation)
```

```
(example:foo/op1 foo-instance 32)
```

(Of course, the specific naming conventions—whether the **operation** was named **example:foo/op1** or **example/foo:op1** or **example.foo:op1**, is orthogonal to the considerations here.)

This mapping has certain clear advantages:

- It is easy to describe and to explain.
- It models the IDL semantics closely.
- It is easy to implement.

- It does not require special method-definition macros—method implementation can be done using standard Common Lisp macros to which it is mapped.

6.15.1.1 *Encapsulation violation*

It forces an invocation of a method on an object to know in precisely which superclass the method was declared. This violates the spirit of object-orientation and obviates certain kinds of polymorphic code.

6.15.1.2 *Ease-of-use problems*

While class names and type names are used fairly infrequently in code, method names are used often, particularly during interactive development. To require use of a fully-scoped name in each case would be cumbersome and inflexible.

6.15.2 *Use of a designated package*

Why not just say

(op1 x 3)

in the above example?

Mostly because it is not clear in what package **op1** should reside.

It cannot be the package named **example** for a similar reason to that in the above case: an interface could inherit an operation from two distinct interfaces in different modules.

Now it seems like we will be OK if we define all operations in the same, fixed, **package**. Suppose we choose a **package** named **IDL**. Then our example becomes:

(idl:op1 x 3)

This is a workable solution, but there are still some problems:

6.15.2.1 *Importing*

We cannot simply import the **idl** package, since the name of an operation may (and often does) conflict with the name of some symbol in the **common-lisp** package.

To require the user to shadow explicitly each conflict seems unwieldy, as the Lisp package system interacts with shadowing in ways that are notoriously counterintuitive.

Thus, this solution requires the explicit prefix of a package before each method invocation. This gets tiresome, particularly in an interactive environment.

6.15.2.2 *Conflict*

This would conflict with user usages of the same package.

6.15.3 *Using a prefix*

The problem of importing, above, can be resolved to some extent by requiring each operation to use a specific character prefix, such as “.” or “/”. While in C++ one might write

```
x.op1(3)
```

in Lisp we would write,

```
(.op1 x 3)
```

where the **.op1** operation was imported from the **idl** package.

The problem with this is that importing symbols in an IDL environment is a procedure fraught with difficulty. The **.op1** symbol would be interned at read time, and if the IDL file declaring the operation had not been loaded correctly, or if it failed to declare the **op1** operation, subtle bugs could arise. Of course it would be possible to check for certain of these bugs, but matters could still get confusing.

6.15.4 *Using the :keyword package*

Since we have seen that dynamic import of symbols generated by the IDL compiler can lead to subtle bugs, why not simply use a simple, well-known package? The canonical such package in Lisp is **:keyword**. This would enable us to write:

```
(:op1 x 3)
```

This seems concise and unambiguous (although we have still not discussed signature issues). Indeed, it is a perfectly reasonable solution. It does have three disadvantages, however:

6.15.4.1 *Appearance*

Some Lisp users have complained that this convention makes a method invocation look too much like a keyword usage.

We also rejected hybrid solutions based on combining keyword mapping with a prefix, as the resulting names looked awkward.

6.15.4.2 *Name conflict*

It could conflict with a user's usage of the **function-value** of a **:keyword** symbol.

6.15.5 Conclusion

We thus were led to the solution we proposed, which allows the very concise `:keyword` usage without prejudicing other names for the package.

6.16 operation mapping: signature

We have chosen our naming convention for operations by process of elimination, but we are not yet done.

It would be most natural to map the operation **op1** into a generic function named **op1** in the operation package.

Thus, we would like to be able to implement the method **op1** on a particular interface via a syntax such as:

```
(defmethod op1 ((receiver example:bar) arg)
  ;...implementation code
)
```

Unfortunately, CLOS mandates the restriction that the methods of a single generic function have congruent lambda lists, which obviates this approach, since it would prevent the implementation of another interface's operation named **op1** that happened to take more than a single parameter.

We could skirt this restriction by defining a metaclass for **op1** different from **standard-generic-function**, but we chose not to rely on the MOP, as stated above.

In practice, we chose a two-pronged approach to the problem.

6.16.1 Leave the signature of the generic function unspecified

This specification does not require any specific signature for the generic function **op1**, but it does require that dispatching be done on the first argument.

6.16.2 Require method definition via a particular macro

Since we want to leave unspecified the particular signature of a method, we must use a macro other than **defmethod** in order to define portably the implementation of an IDL operation on the server side.

As a first draft, we chose the syntax to be reasonably close to **defmethod**.

However, we intend to consider several changes in the final version. In order to avoid problems with loading, we designed the macro not to make use of information in the interface repository at this time.

6.16.2.1 **any** handling

We might want to specify whether any parameters are to be unwrapped automatically.

6.16.2.2 Handling of **out** and **inout** parameters

We might want to define a syntax for automatic handling of **out** and **inout** parameters, although we doubt that this is worth the trouble.

6.16.2.3 Integration with class definition macro

Most ambitiously, we could define a macro that combined the functionality of **defclass** and **corba:define-method** to allow definition of methods in a way similar to Java or C++; this is closer to the semantic model of IDL. Here, methods would be defined within the scope of some class-definition macro. This could lead to simpler local slot or attribute accessors, and a particular variable, such as **this** or **self**, could be bound to the receiver.

6.17 **operation** mapping: parameter passing modes

The major issue here was whether to attempt to simulate explicitly the semantics of the IDL call-by-value-result for **inout** parameters.

For example, we could have taken an approach analogous to the **Holder** classes of Java and C++.

In practice, however, these features are rarely used and are adequately—and much more simply—handled by using the Lisp multiple values mechanism. Also, the semantics of **inout** parameters are not canonical in the presence of nested constructed types.

We considered also mapping **operation** into an auxiliary macro that handled the multiple-value bookkeeping automatically. We rejected this approach for this draft because the semantics of **inout** parameters in their interaction with constructed types did not seem clear to us.

6.18 Mapping of **attribute**

Many of the considerations for naming **attribute** are similar to those for naming **operation**. We chose a naming convention similar to that for **operation**.

We assume as a design parameter that the writer function for an **attribute** will be formed by treating the reader form as a generalized place for the purposes of **self**.

6.19 *Compiler mapping*

Languages which lack first-class access to their compiler typically standardize only the run-time environment and leave the IDL compilation unstandardized. The IDL compiler is usually implemented as a separate program whose interface is defined by the ORB vendor.

We considered two compiler interfaces: the current one and an interface that decoupled the parsing and the compilation. The parse interface would simply build an interface repository from the source file; the compilation interface would compile from an interface repository. However, the current mapping is much simpler.

6.20 *Pseudo Interface Mapping*

The main question in mapping the pseudo-interfaces was whether to use Lisp conventions throughout or simply translate the pseudo-IDL in “brute-force” fashion.

We chose the latter approach for two reasons:

- It’s easier.
- A high-quality Java pseudo-interface mapping already exists.

The Java pseudo-interface mapping already does much of the work of smoothing out the rough edges of the raw IDL (particularly in the DII) so we stayed very close to their mapping. We feel this approach will also reduce the learning curve for users familiar with Java (or C++, upon which the Java mapping in turn was based).

However, our treatment of **any** was chosen to simplify use of the DII, since Lisp values can be used directly.

6.21 *Server side mapping*

One of the most interesting issues here was whether to allocate slots automatically based on interface attributes. On the positive side, doing so significantly simplifies common usages and examples. On the negative side, it is unnecessary in certain cases.

