Lisp Mapping Specification

New Edition: May 2000

Contents

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBAservices: Common Object Services Specification* contains specifications for OMG's Object Services.

The OMG collects information for each specification by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- BBN
- Boeing
- Franz, Inc.
- Fujitsu
- IBM
- MITRE
- Raytheon

Overview 1

What color is a chameleon on a mirror?

-- *Steve Wright (attributed)*

Contents

The contents of this chapter are not normative. This chapter contains the following sections.

Section Title	**Page**
“Introduction”	1-1
“Why is the Lisp Mapping Difficult?”	1-2
“Mapping Goals”	1-2
“Mapping Principles”	1-3
“Expressing the Mapping in UML”	1-4

1.1 Introduction

The purpose of this chapter is to describe fundamental design principles expressed in the actual mapping.

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.

1.2 Why is the Lisp Mapping Difficult?

The reason the Lisp mapping is difficult, and the reason that so much effort is expended here and in Appendix A to justify the mapping, is precisely that, in some sense, the mapping is too easy.

Many Lisp programmers will idiomatically begin the development of an application solution in a domain by tailoring the Lisp language itself to that domain.

Indeed, Lisp offers a plethora, if not a surfeit, of features to support this: reader macros, meta-object protocol, macros, compiler macros, code-generation, on-the-fly-compilation, mutable objects and classes, and so on.

In the particular case of mapping into IDL, however, the constraint is a bit more complex, since an IDL mapping should feel idiomatic to Lisp programmers. Thus, the design of a Lisp mapping is invariably driven by the tension between making the syntax and semantics as IDL-friendly as possible and making the syntax and semantics as friendly as possible, not so much to the Lisp language per se, but to some common informal application-independent idioms used by Lisp programmers.

This mapping design thus represents a continual resolution of the tension between these two idioms: the adaptation of Lisp to CORBA, which is one type of Lisp idiom, and the adaptation of CORBA to Lisp.

For example, consider the simplest possible mapping: that of identifiers. We reviewed at least ten different identifier mappings, many of which were implemented in prototype ORBs. For example, some Lisp programmers like to notate identifiers by separating words with a hyphen, while in IDL, case changes or underscores are idiomatic. Of course, Lisp can fully support the IDL style, but should it? Answers vary. (We chose to use IDL-style identifiers).

This simple tension was expressed on many levels throughout the mapping. In order to resolve it satisfactorily but not in an ad hoc way, we formulated a set of design principles, and we exhaustively prototyped and commercially implemented the various recommendations. We eventually were able to distill the mapping principles to a small set of natural mapping principles which informed and motivated the specific mapping. The UML metamodel has proven extremely useful in expositing these principles.

1.3 Mapping Goals

The five design goals of this mapping are:

1. Ease of use
2. Consistency
3. Flexibility

4. Performance
5. Adherence to IDL

1.3.1 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 generate 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.

1.3.2 Consistency

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. Consistency was achieved by viewing the mapping as a mapping on the UML metamodel underlying the IDL type model.

1.3.3 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.

1.3.4 Performance

The features described here should not impose undue performance overhead.

1.3.5 Adherence to IDL

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

1.4 Mapping Principles

The key goal in the mapping design was to designate a small set of principles that could be uniformly applied to generate the mapping.

Our motivation, thus, was to produce a mapping that is not only useful, but is also easy-to-learn and aesthetically pleasing.

In specifying and in implementing a language mapping, it is difficult to avoid close attention to the details of IDL language and target language syntax. Mapping documents sometimes loosely speak of mapping from IDL syntactic elements to target language syntactic elements.

We used the UML meta-model informally to express the key concepts in our mapping independent of the syntactic specifics of their expression in IDL.

Note – Although the principles underlying the mapping are expressed informally in UML, familiarity with the UML metamodel is not necessary in order understand these principles. In fact, it normally suffices simply to keep in mind standard object-oriented terminology and praxis; the UML metamodel serves primarily only as a way to exposit, but not necessarily to design, the mapping principles.

1.5 Expressing the Mapping in UML

Our mapping is normatively specified by its action on textual entities: the input IDL and the output Lisp. Although precise, this prescription has the flaw that it obfuscates the underlying principles used to select these textual representations. The mapping is actually more clearly thought of directly as a transformation on UML models. The mapping on models is much simpler and crisper than the mapping on sources.

Thus, the expression of the underlying principles governing our choice of mapping can be clearly and succinctly expressed using the UML metamodel. (The presentation here is informal and purposely elides some details, of course.)ping. The UML metamodel. (The presentation here is informal and purposely elides some details, of course.)

Because the native Lisp object model is farther from the IDL object model than is the case with Java and C++, it is useful in discussing the Lisp mapping to describe more generally the nature of a language mapping.

1.5.1 UML Metamodel

We now present some salient aspects of the UML 1.3 meta-model that will be used in the description in this section:

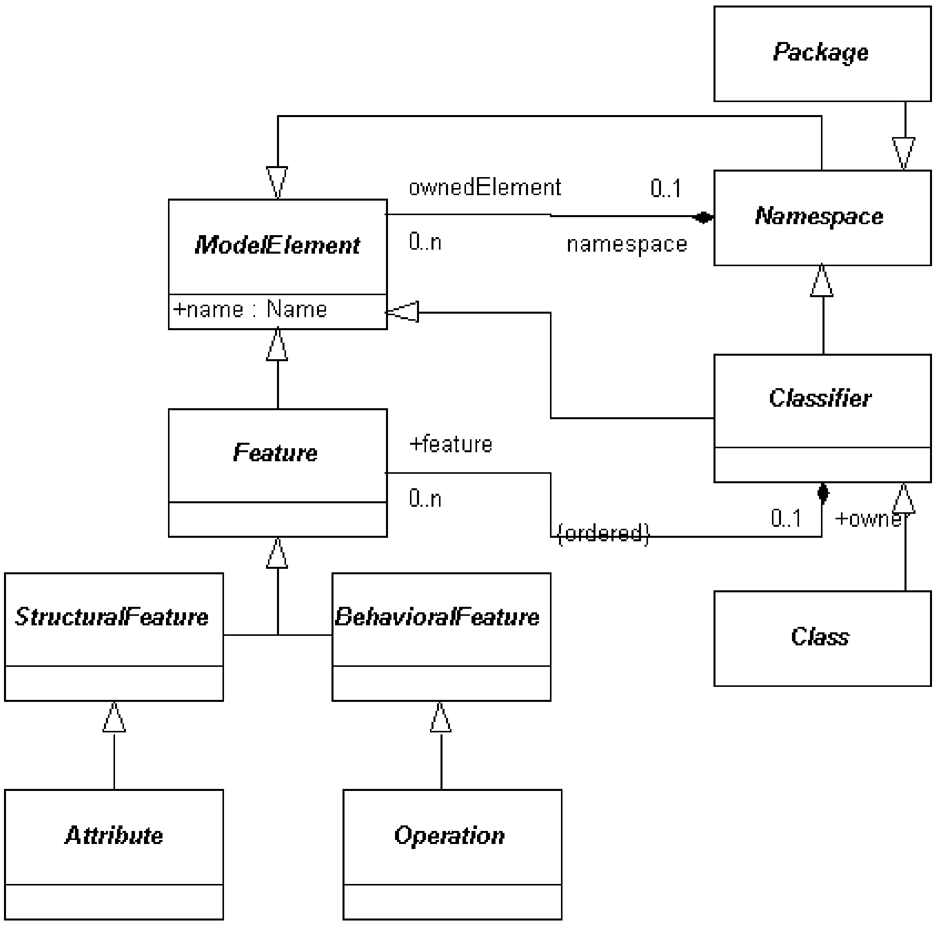

Figure 1-1 Simplified UML 1.3 meta-model

In a static class diagram representing a model of an IDL file, each struct, union, interface, exception, and valuetype defined in that file will correspond to a **Classifier** object.

A UML model comprises various model elements.

- A *Classifier* is a kind of model element that describes behavioral and structural features. We view typedefs, exceptions, structs, interfaces, and so on as defining Classifier elements in the model.
- A *Feature* is a property encapsulated in a Classifier. A *StructuralFeature* refers to static model elements; we consider IDL attribute and member declarations to correspond to instances of StructuralFeature. We consider IDL operation declarations to correspond to instances of Operation. The Operation and StructuralFeature classes have, of course, other attributes for determining data type, calling sequence, and so on; these are not shown in this diagram.

- A *Namespace* holds a collection of model elements; each classifier, but most importantly the *Package*, is a Namespace.

1.5.2 UML Overview of a Mapping

A mapping is fundamentally a transformation of object models: As input it takes a model representing a set of IDL files; as output it produces a model representing a set of Lisp data types. This is illustrated informally in the figure below.

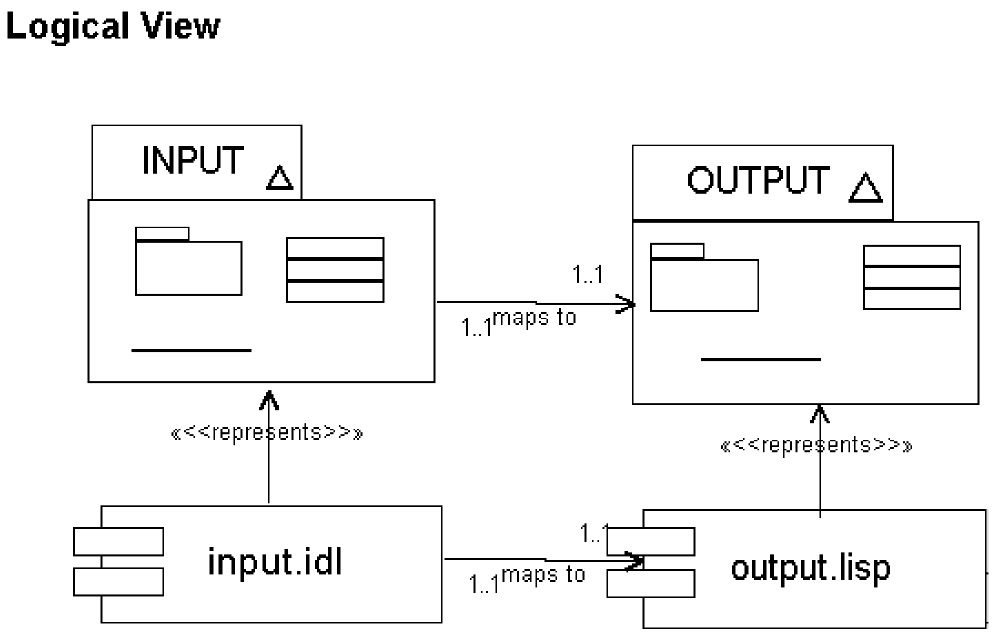

Figure 1-2 Language mapping from IDL model to Lisp model

Figure 1-2 represents the conceptual entities involved in a Lisp language mapping. Each entity is described below:

1.5.2.1 input.idl

The *input.idl* file represents a source file in IDL. The goal of the language mapping is to determine how the datatypes defined in this file may be accessed from the target language, in this case Lisp.

1.5.2.2 INPUT

The *INPUT* system model is intended here to represent a static UML class diagram which represents the elements defined in input.idl. Because the UML Profile for CORBA is not standardized at the time of this writing, some common-sense guesses on the appropriate UML representation for IDL elements may be made; these do not impact the underlying mapping.

For example, we can assume that each IDL data type is a Classifier; that IDL interfaces are represented by Class objects, that interface inheritance is represented by a Generalization association, that modules are represented by Package, and so on.

At any rate, the INPUT model is a static model of the input IDL file: it is a normal UML class diagram representing each element of the IDL file.

1.5.2.3 OUTPUT

Each model element - packages, classes, associations, and so on - in the input model, which corresponds to the file input.idl, is transformed by the mapping into a set of model elements in the output model. In fact, for the most part the input and output models are identical. **Association** maps to **Association**; **Classifier** maps to **Classifier**, **Package** maps to **Package**, **Operation** maps to **Operation**. There is slight complexity in that an interface class in the input model maps to several classes in the output model (stub and skeleton classes); similarly, a Generalization between two interface Class objects in the input model is mapped to several Generalizations in the output model.

1.5.2.4 output.lisp

Once output model is generated, the Lisp datatypes, which we here informally designate by the source file "output.lisp" is generated. Lisp classes and data types are defined by the mapping whose model is the OUTPUT model.

1.5.3 What a Mapping Needs to Specify

We have seen that a language mapping needs to specify two parameters: Generation of the output model from the input model; and “forward-engineering” of the target language from the output model.

There is not much to say about generation of the output model from the input model. This part of the mapping is essentially language-independent, has been done numerous times in other mappings, and is well-understood. It is the code-generation of Lisp from the OUTPUT model where the design choices become more important, and it is this point that we now address.

1.5.4 Generation of Lisp from UML

We have reduced the unstructured problem of mapping the multitudinous and multifarious elements of IDL syntax and semantics into the more structured problem of mapping (some) UML constructs into Lisp.

Because these UML constructs themselves already represent instances of a simple and clean meta-model, the UML 1.3 meta-model, we are led to the following observation:

In order to determine a mapping from IDL to Lisp, it suffices to determine a mapping for each element of the UML metamodel.

Once phrased in this way, the correct choice of mapping becomes clearer.

For example, both IDL interface and IDL module are namespaces. One early Lisp mapping mapped both module and interface to the Lisp package—it may not be immediately obvious whether interface should map to Package.

On the other hand, it is in fact rather obvious that the UML **Package** element should map to Lisp **package**. Since it is equally clear that IDL **module** (and not IDL **interface**) map to UML **Package**, we infer quite naturally that IDL **module** maps to Lisp **package**.

In fact, as it turns out the mapping of each UML meta-model element is actually quite natural. The following table outlines this mapping.

Table 1-1 UML metamodel element mapping

UML metamodel element	**Lisp mapping**
Classifier Namespace	type naming prefix
Package Name	package symbol
Class generalization	class inheritance
OwnedElement StructuralFeature	/ slot
BehavioralFeature	method

1.5.4.1 *Mapping for Namespace*

A top-level namespace is named by the symbol that is its own name. Otherwise, the namespace is owned by some parent namespace; the name of the namespace is in this case the concatenation of the name of the parent with the mapping for ownedElement (the ’/’ character) with the name of the namespace.

1.5.4.2 *Mapping for Package*

The simplest namespace is Package. This is simply mapped to the Lisp **package**.

When this class diagram is mapped into Lisp, the **Classifier** objects are unchanged.

Hence, we have reduced the problem of mapping the various disparate IDL datatypes with their corresponding disparate syntax into the much more uniform problem of mapping a **Classifier** object into Lisp.

Since **Classifier**s have**Feature**s, we also much map the **Feature** data type.

Our mapping boils down, therefore, to the following rules:

1.5.4.3 Mapping for Feature

- A **StructuralFeature** is mapped to a slot. This slot has accessors that follow standard Lisp conventions. Specifically, each such slot corresponds to an initarg whose name is the name of the slot and whose package is **:KEYWORD**, and an accessor function whose name is the name of the slot.
- A **BehavioralFeature** is mapped to a method. The first actual parameter of this method is the target of the operation.
- The Lisp name of the Lisp entity corresponding to a feature is the symbol whose name is the name of the **Feature** and whose package is the **Feature** package.

1.5.4.4 Mapping for Namespace

A **Namespace** is corresponds to a symbol formed as follows. Concatenate the names of all the namespaces containing the given Namespace, outermost to innermost, separated by the "/" character. Change the "/" preceding the first Namespace that is not a Package to ":". This name the symbol corresponding to that Namespace.

We can think of this mapping as: “elementOwnership” maps to “/” is we like.

1.5.4.5 Mapping for Classifier

A Class is mapped to a Lisp class that inherits directly from the Lisp class corresponding to its parents. The root of this inheritance hierarchy is the class **CORBA**:<name> where <name> is one of **UNION**, **STRUCT**, **VALUETYPE**, **OBJECT**, **EXCEPTION**, **ABSTRACTBASE**. This Lisp class has slots and methods corresponding to the **Feature**s of the**Classifier**.

Each such **Class** has a constructor whose name is the name of the **Class** and which takes keyword initialization arguments given by its **StructuralFeature**s.

An **interface Class** has some auxiliary classes generated.

If a **Classifier** is a not a **Class** then it is a **typedef** and there is no inheritance or Feature mapping.

1.5.4.6 Example:

Consider the following simple UML diagram:

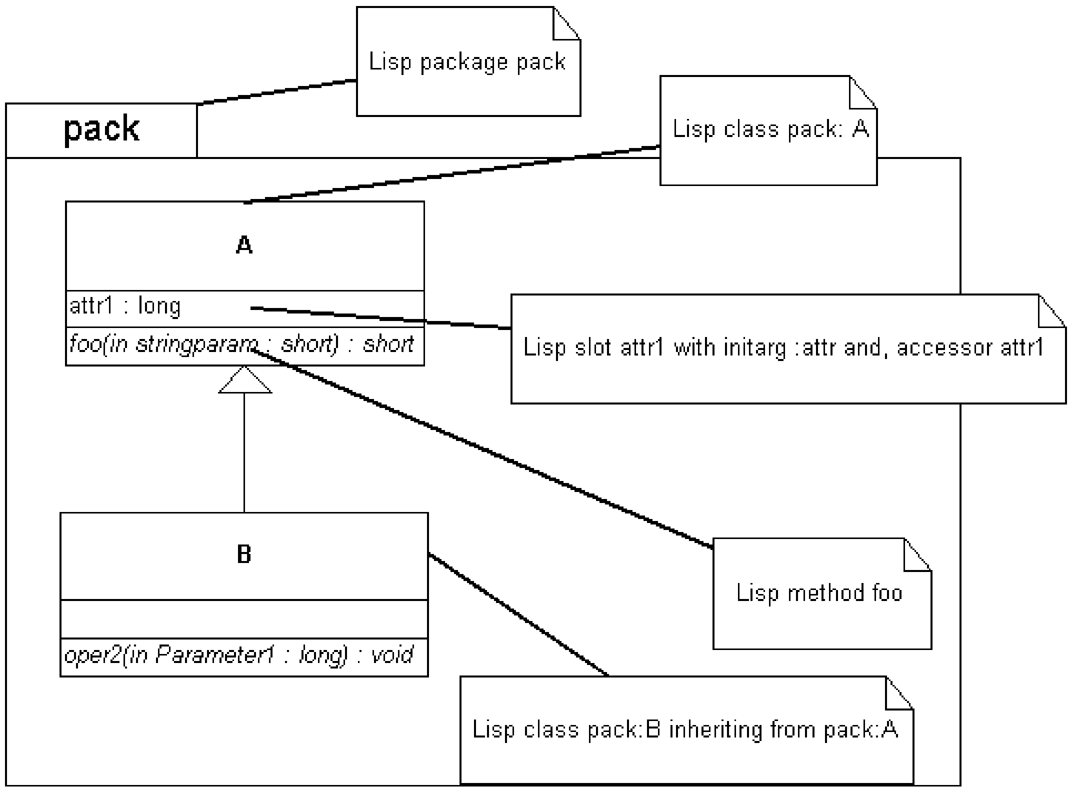
Lisp package pack
pack
Lisp class pack: A
A
attr1 : long
foo(in stringparam : short) : short
Lisp slot attr1 with initarg :attr and, accessor attr1
B
oper2(in Parameter1 : long) : void
Lisp method foo
Lisp class pack:B inheriting from pack:A

Figure 1-3 Sample UML diagram to be mapped into Lisp

The **pack** Package element corresponds to a Lisp package named **pack**. The **A** Classifier corresponds to a Lisp class. Since **pack** owns **A**, **A** is named, in Lisp, **pack:A**.

Note that if **A** were defined in the package **pack2** which itself was included in **pack**, then **A** would map to a Lisp class named **pack/pack2:A**.

A has two Features with names **attr1** and **oper1**. These correspond to elements of the **Feature** package in Lisp.

There is a Lisp class named **pack:B** which inherits from **pack:A**. It adds an operation named **oper2**.

1.5.5 Invocation and Definition

Implicit in the UML meta-model is that there shall be a way:

- To access a Feature
- To invoke an operation
- To implement a method

The key here is this:

Feature operations shall be independent, to the extent possible, of the stereotype of the associated class.

For example, suppose object **x** has a feature named **“foo”**. This rule states:

The value of the feature **foo**, and the means for accessing this value are independent of whether **x** happens to be a union, a struct, a valuetype, an abstract interface, an exception, an interface, or a valuetype.

In point of fact, in this case, the feature is always accessed via that form:
`(foo x)`

The value of the feature is always written via form like:
`(setf (foo x) 3)`

Here, the symbol “foo” is in the Feature package, with nickname :OP, so this could be written **(op:foo x**) and so on.

For example, this rule is one of the key reasons we did not force the metaclass of mapped struct to be instances of Lisp structure-class, which use a different syntax for accessing members.

Similarly, if **foo** were an Operation with parameters a, b, c..., invocation is always via **`(op:foo x a b c).`**

The invocation mapping is summarized in the figure below.

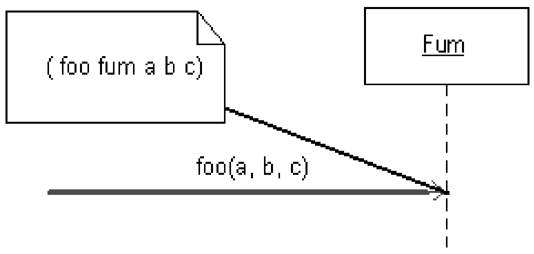

Figure 1-4 Sending message foo is mapped to invocation of method foo with parameters the target object and each of the actual parameters.

1.5.6 Pseudo IDL

A similar system could be drawn for the various PIDL elements. The mapping for PIDL is similar, in any case, to that for IDL.

1.5.7 Additional information

Appendix A outlines some of the issues involved more specifically, as well as discussing thorny but necessary matters such as character sets, name collisions, and the like.

Mapping IDL to Lisp 2

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

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.

Contents

This chapter contains the following sections.

Section Title	**Page**
“Mapping Concepts”	2-2
“Semantics of Type Mapping”	2-2
“Mapping for Basic Types”	2-3
“Introduction to Named Types”	2-6
“Distinguished Packages”	2-7
“Scoped Names and Scoped Symbols”	2-8
“The package_prefix pragma”	2-9
“Mapping for Interface”	2-10
“Mapping for Operation”	2-12
“Mapping for Attribute”	2-14
“Mapping of Module”	2-15
“Mapping for enum”	2-16
“Mapping for Struct”	2-17

2.1 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, generated by the mapping.

2.2 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 is ***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 a 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** shall be signaled.

2.3 Mapping for Basic Types

2.3.1 Overview

Table 2-1 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:

1. The entry “Name of Lisp type” is a symbol that names the type represented by this set of Lisp values.
2. The entry “Lisp type specifier” is a standard Common Lisp type specifier that denotes this set of Lisp values.

Table 2-1 Basic Type Mappings

IDL Type	Name of Lisp Type	Lisp Type Specifier
boolean	corba:boolean	*generalized 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:unsigned short	(unsigned byte 16)
long	corba:long	(signed byte 32)
unsigned long	corba:unsigned long	(unsigned byte 32)
long long	corba:longlong	(signed byte 64)
unsigned long long	corba:ulonglong	(unsigned byte 64)
float	corba:float	see text

Table 2-1 Basic Type Mappings

IDL Type	Name of Lisp Type	Lisp Type Specifier
double	corba:double	see text
long double	corba:longdouble	see text
fixed	corba:fixed	see text

Additional details are described in the following sections.

2.3.1.1 Example

```
(typep -3 ‘corba:short)
> T
(typep -3 ‘corba:ushort)
> nil
(typep “A string” ‘corba:string)
> T
```

2.3.2 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 the type T, also called generalized boolean.

2.3.3 Char

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

2.3.3.1 Usage example

```
(typep #\x ‘corba:char)
> T
(typep “x” ‘corba:char)
> nil
```

2.3.4 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**).

2.3.4.1 Usage example

```
(typep 255 ‘corba:octet)
> T
(typep -1 ‘corba:octet)
> nil
```

2.3.5 Wchar, Wstring

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

2.3.6 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**.

2.3.6.1 Usage example

```
(typep “A string” ‘corba:string)
> T
(typep nil ‘corba:string)
> nil
```

2.3.7 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**.

2.3.8 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 shall be subtypes of the type **real**. They shall allow representation of all numbers specified by the corresponding CORBA types.

2.3.9 Fixed

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

2.4 Introduction to Named Types

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

2.4.1 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.

2.4.1.1 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.

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**.

2.4.1.2 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.”

2.5 Distinguished Packages

This document will refer to two kinds of packages:

1. A package that comprises those packages defined explicitly by this specification.
2. A package that comprises those packages created as a result of compiling user IDL code.

The first kind of package consists of these three distinct packages: the ***root package***, the ***corba package***, and the ***Feature 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 Feature package is the string “OMG.ORG/FEATURE”.

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 Feature 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.”

2.5.1 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 shall support the following default nicknames:

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

This document will use these nicknames without comment.

2.6 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
- valuetype
- abstract interface
- 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.

2.6.1 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.

2.6.1.1 Examples of scoping symbols

Consider the following 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.”

Here is a more complex example of 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**.

Finally, another example:

```
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**.

2.7 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 within its scope as follows: the name of the scoping symbol for such a top-level module within the scope of a package_prefix pragma is the concatenation of the given **package_prefix** with the name of the module. The scope of the package_prefix pragma follows the same rules as the scope of the prefix pragma defined in *The Interface Repository* chapter of the CORBA Core specification.

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.

2.7.1 Example

2.7.1.1 IDL

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

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

2.8 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 *Server Side* mapping chapter of this specification.

2.8.1 Example

2.8.1.1 IDL

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

2.8.1.2 generated Lisp

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

2.8.2 Stub classes

An IDL interface named I generates a stub class whose name is the concatenation of the name of the scoped symbol for I to the string "-PROXY" and whose package is the package of the scoped symbol for I.

The direct superclasses of the -PROXY class corresponding to an interface I are determined as follows. If I has no declared base interfaces, the generated class has the direct superclasses, the Lisp class corresponding to interface I, and the class corba:proxy. Otherwise the generated class has as direct superclasses the Lisp class corresponding to interface I and the-PROXY classes corresponding to each of the declared base interfaces of I.

2.8.2.1 Example

The stub classes generated for the IDL above are:

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

The IDL and the Lisp in the example is represented non-normatively in UML in the pair of figures below.

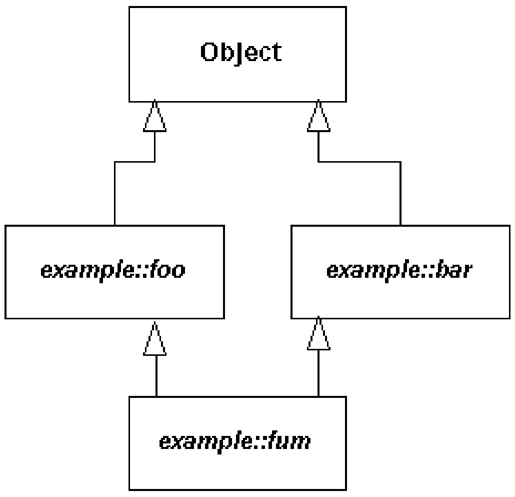

Figure 2-2 Non-normative UML for example 2.8.2.1, IDL

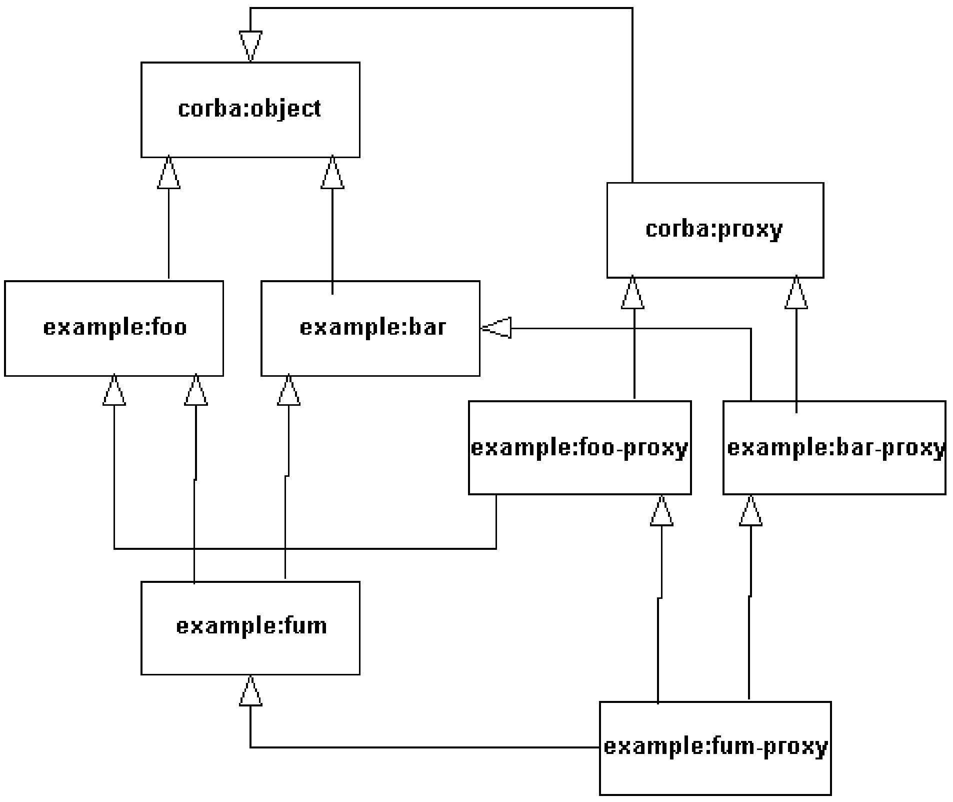

Figure 2-3 Non-normative UML for example 2.8.2.1, IDL

2.9 *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 Side* mapping chapter of this specification. The contents of this section apply to operations declared within **interface**s, **abstract interface**s, and **valuetype**s.

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 Feature package.

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

2.9.1 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.

2.9.2 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.

2.9.3 One-way

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

2.9.4 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.”

2.9.5 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.

2.9.6 Context

For each context name declared by an operation, the operation accepts a single additional argument whose type is a first-class context, accessed via the standard **Context** accessors.

2.9.7 Example

2.9.7.1 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);
};
```

2.9.7.2 generated Lisp

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

2.9.7.3 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”)
```

2.10 Mapping for Attribute

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

2.10.1 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 Feature package.

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

2.10.2 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 attribute**s. A writer is defined whose name is **(setf name)** where **name** is the name of the defined reader-method.

2.10.3 Example

2.10.3.1 IDL

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

2.10.3.2 Usage

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

2.11 Mapping of Module

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

2.11.1 Example

2.11.1.1 IDL

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

2.11.1.2 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 ...)
```

2.12 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.

2.12.1 Example

2.12.1.1 IDL

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

2.12.1.2 generated Lisp

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

2.12.1.3 usage

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

2.13 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 Feature 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.

2.13.1 Example

2.13.1.1 IDL

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

2.13.1.2 generated Lisp

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

2.13.1.3 usage

```
(setq struct (structmodule:struct_type
:field1 100000
:field2 “The value of field2”))
(field1 struct)
> 100000
(setf (field1 struct) -500)
(field1 struct)
> -500
```

2.14 Mapping for Union

An IDL **union** is mapped to a Lisp **class** named by 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 Feature package and an initialization argument named **:union-discriminator**.

The value can be accessed using the accessor function named **union-value** in the Feature 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**.

2.14.1 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.

2.14.2 Example

2.14.2.1 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;
}; };
```

2.14.2.2 generated Lisp

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

2.14.2.3 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
```

2.15 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.

2.15.1 Example

2.15.1.1 IDL

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

2.15.1.2 Generated Lisp

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

2.16 Mapping for array

An IDL **array** is mapped to a Lisp **array** of the same rank. The element type of the mapped **array** shall 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.

2.16.1 Example

2.16.1.1 IDL

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

2.16.1.2 Generated Lisp

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

2.16.1.3 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
```

2.17 Mapping for sequence

An IDL **sequence** is mapped to a Lisp **sequence**. Bounds checking shall be done on bounded **sequence**s when they are marshaled as parameters to IDL operations and an IDL CORBA::MARSHAL exception shall be raised if necessary.

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**.

Note – 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.

2.17.1 Example

2.17.1.1 IDL

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

2.17.1.2 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
```

2.18 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.**

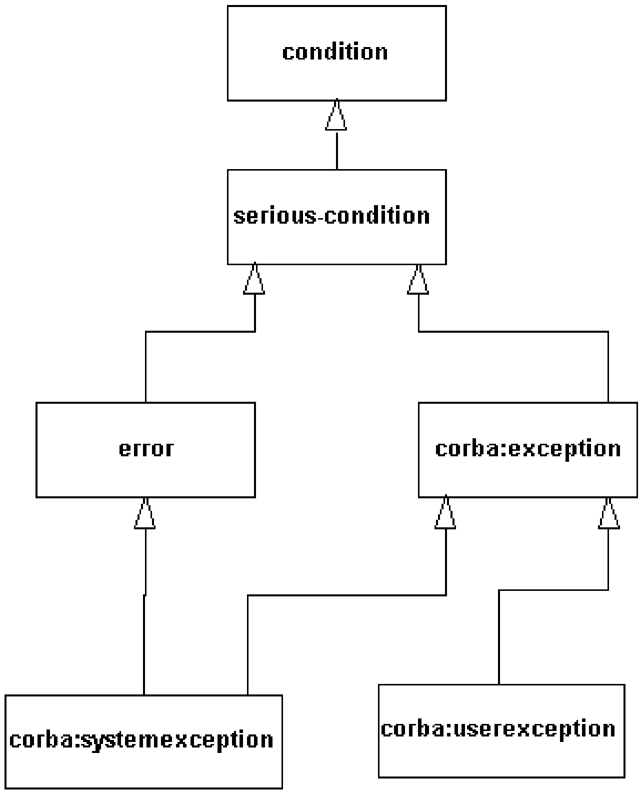

Figure 2-3 Lisp condition hierarchy

System exceptions inherit from a condition named **corba:systemexception,** which also inherits from the condition **error**.

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

2.18.1 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**.

2.18.1.1 Example

Consider the following IDL:

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

The Lisp corresponding to this fragment might look like the following:

```
(defpackage :example)
(define-condition example:ex1 (corba:userexception)
((reason :initarg :reason ...))
; Usage example
(error (example:ex1 :reason “Example of condition”))
```

2.18.2 System Exception

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

2.19 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**.

2.19.1 Example

2.19.1.1 IDL

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

2.19.1.2 generated Lisp

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

2.19.1.3 Usage example

```
(typep -3 ‘example:foo)
> nil
(typep 6000 ‘example:bar)
> nil
(typep “hello” ‘example:bar)
> T
```

2.20 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.

2.20.1 Constructors

The constructor **corba:any** takes two keyword arguments named **any-value** and **any-typecode**. If **any-typecode** is specified, then **any-value** shall be specified. If **any-value** and **any-typecode** are each specified, then **any-value** shall 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).

2.20.2 Typecode accessor

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

IF	THEN ....
v is a **valuetype**	the default coercion rules specified below may be overridden by the ORB.
v was created by an invocation of **corba:any**	the actual typecode of **v** is the **any-typecode** argument supplied to **corba:any**.
v is an integer	the actual typecode of **v** is the typecode of the smallest integer type that of which **v** is an instance. Specifically if **v** is of type **corba:unsignedlonglong** or **corba:longlong**, then the actual typecode of **v** is the typecode that describes the first Lisp type among (**corba:short, corba:ushort, corba:long, corba:ulong, corba:longlong, corba:ulonglong**) of which **v** is a member.
	Otherwise if **v** is a member of **corba:float**, **corba:double,** or **corba:longdouble** then the actual typecode of **v** is **corba:tc_floa**t or**corba:tc_double** or **corba:tc_longdouble** respectively.
	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 a boolean then the actual typecode of v is **corba:tc_boolean**.

	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 actual typecode of **v** is a typecode describing a **sequence** compatible with the contents of **v**.
	Otherwise if **v** is an instance of **corba:object**, **corba:struct**, **corba:valuebase**,or **corba:union**, then the actual typecode is the typecode describing the **interface**, **struct**, **valuetype**,or **union** of which **v** is an instance. (Such a **v** is said to be *self-typing*).

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

2.20.3 value accessor

If **v** is a number, a string, a sequence, a boolean, or an instance of **corba:enum**, **corba:object**, **corba:valuetype, corba:struct**, or valuetype, then **(corba: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 **(corba: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.

2.20.4 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

```
(op: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

```
(op: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**.

2.20.5 Additional examples of any usage

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

2.21 Mapping for valuetype

An IDL **valuetype** is mapped to a Lisp class whose name is the scoped symbol for that type.

2.21.1 Inheritance of valuteype

If a valuetype **A** inherits from a valuetype **B**, the generated Lisp class for **A** shall be a subclass of the generated Lisp class for **B**.

2.21.1.1 Example

The IDL

```
module example{
valuetype b {};
valuetype a : b {};
};
```

corresponds to the Lisp classes:

```
(defclass example:b(corba:ValueBase)())
(defclass example:a (example:b)())
```

2.21.2 Valuetypes supporting interfaces

If a valuetype **A** supports an interface (or abstract interface) **B**, then the generated Lisp class for **A** shall be a subclass of the generated Lisp class for **B**.

2.21.2.1 Example

The IDL

```
module example {
interface I {};
valuetype a supports I {};};
```

corresponds to the Lisp:

```
(defclass example:I (corba:Object)())
(defclass a (corba:ValueBase example:I)())
```

2.21.3 Base class for valuetype

If a valuetype **A** is not declared to inherit from any other valuetype **B**, then the generated Lisp class corresponding to **A** shall be a subtype of the Lisp class named **`corba:ValueBase`**.

2.21.3.1 Example

The IDL:

```
module example {
valuetype foo (){};
};
```

corresponds to the Lisp:

```
(defclass example:foo (corba:ValueBase)())
```

2.21.4 Valuetype members

The mapping for valuetype members is based on the mapping for StructuralFeature.

Each member of the given valuetype is mapped to a slot whose name is the symbol whose print-name is the uppercased name of the member and whose package is the Feature package. This slot has an associated initializer keyword whose name is the uppercased name of the member and whose package is the keyword package.

Each member of the **valuetype** is in addition mapped to a pair of accessors of the generated class using the same naming convention as for struct members: a reader whose name is the name of the associated slot and whose package is the Feature package and a writer whose name is the list whose **car** is the symbol **setf** and whose **cdr** is the symbol that names the associated slot.

2.21.4.1 *Example*

Consider the IDL:

```
module example {
valuetype A {
long long b;
short c;
private string d;};};
```

If x is bound to an instance of example:a, then an invocation sequence might be:

```
(op:b x)
---> -400000000
(setf (op:b x) 500000000000000)
(op:b x )
---> 500000000000000
(op:c x)
---> -100
(slot-value x 'op:x)
--->-100
(slot-value x 'op:d)
"Sample private member value"
```

2.21.5 *Valuetype operations*

An operation declared within a valuetype is mapped to a Lisp function using the same naming convention as for operations declared on interfaces.

An operation may be implemented using the **corba:define-method** macro.

2.21.5.1 *Example*

Consider the following IDL:

```
module M {
valuetype A {
long foo (in string s);};
```

If x is bound to an instance of class M:A, then the following is a sample invocation sequence:

```
(op:foo x "input")
---> -200
```

2.21.6 Boxed values

Suppose **B** is the name of an IDL valuetype that is a boxed value for an IDL type **M**. If **M** is a primitive type, then **B** corresponds to a Lisp type whose name is the scoped symbol for **B** and which holds a single member named **data**. Otherwise **B** is mapped to the type whose name is the scoped symbol for **B** and which denotes the type of the Lisp type corresponding to **M**.

2.21.7 Value factory

The IDL native type **CORBA::ValueFactory** maps to the Lisp class **class**.

Each implementation of a valuetype is associated with a keyword defined on the **shared-initialize** method for that class:

- If the value of the **:create-for-unmarshal** keyword is non-**nil**, the **shared-initialize** method is to be executed in the context of unmarshalling an instance of that type by the ORB; this corresponds to the **create_for_unmarshal** pseudo-operation. The value of this keyword shall be set to non-**nil** only by the ORB: users may not portably invoke **shared-initialize** with non-**nil** value of this keyword parameter.
- Otherwise, if the value of the **:factory** keyword is non-**nil** it shall be a symbol whose print-name is the (uppercased) name of an initializer for that valuetype. The parameters of the initializer are specified as the values of the keyword parameters whose print-names correspond to the uppercased names of the names of those parameters.
- Otherwise, if the value of the **:factory** keyword is **nil** or if it is unbound, then the remaining keyword initializers are treated as slot initializers (which are defined by the mapping for the valuetype's declaration) or as other user-defined keyword initializers.

Each non-**abstract** valuetype is associated with a default constructor whose name is the name of that symbol and which signals an **error** if the value of the **:factory** keyword parameter is non-**nil**.

2.21.7.1 Examples

Consider the following IDL:

```
module example {
valuetype A {
string bar;
boolean fum;
factory c (in long x);
};};
```

An instance of this valuetype may be created via the call

```
(make-instance ’example:A :bar "hello" :fum T)
```

A user class B can provide a factory implementation by specifying the behavior of forms like:

```
(make-instance ’B :factory ’c :x 898)
```

2.21.8 Unmarshalling Issues

When the ORB unmarshals a valuetype for a request, it tries to find the class that corresponds to that valuetype via the **ORB::lookup_value_factory** operation. If the factory lookup succeeds, the instance is instantiated by invoking the constructor associated with that factory.

If the factory lookup fails, then if the repository ID begins with string **IDL:**, the associated factory is that corresponding to the symbol that names the valuetype whose generated repository ID is the same as that repository ID. If such a valuetype class exists and is not **abstract**, an instance of that class is unmarshalled using the default factory for that valuetype class.

Otherwise the CORBA::MARSHAL exception is signalled back to the client.

2.21.9 Mapping for Abstract Valuetypes

The Lisp class corresponding to an abstract valuetype **B** is the class corresponding to the valuetype **B’** in the IDL formed from the original IDL definition of B by removing the **abstract** specifier. It is an error if the user directly instantiates such a class.

2.21.9.1 Mapping for abstract interface

The mapping for **abstract interface** is the same as the mapping for **interface** except that each abstract interface inherits from the class **corba:abstractbase**, the mapping for the native type **CORBA::AbstractBase**. Neither generated servant nor proxy classes inherit from **CORBA:AbstractBase** however. The class **corba:abstractbase** inherits from **corba:Object**.

When used as the declared base class of an interface declaration, the mapping for that interface is treated exactly as if the abstract interface were an interface.

Similarly, when a valuetype is declared to support an abstract interface, the abstract interface is treated as an interface.

It is an error if the user directly instantiates such a class.

2.21.10 Example

2.21.10.1 IDL

```
module example{
abstract interface foo{};
abstract interface bar {};
abstract interface fum : foo, bar {};
interface c :fum {};
valuetype d supports fum{};
```

2.21.10.2 Generated Lisp

```
(defclass example:foo(CORBA:AbstractBase)())
(defclass example:bar(CORBA:AbstractBase)())
(defclass example:fum (example:foo example:bar)())
(defclass example:c (example:fum))
(defclass example:d (corba:valuebase example:fum-servant))
```

2.22 Custom Valuetypes

Valuetypes declared as **custom** shall inherit from the class **CORBA:customMarshal**. This class, and the associated **DataOutputStream** and **DataInputStream** classes, are mapped according to their definition in OMG IDL. The user implementation shall implement the **op:marshal** and **op:unmarshal** operations declared in **CORBA:customMarshal** in order to implement custom marshalling and unmarshalling.

Mapping Pseudo-Objects to Lisp 3

Contents

This chapter contains the following sections.

3.1 Introduction

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**.

This mapping of the pseudo objects was modeled after that in the Java mapping.

3.1.1 Pseudo Interface

The use of **pseudo interface** is a convenient device, which means that most of the standard language mapping rules defined in this specification may be mechanically used to generate the corresponding Lisp values. However, in general the resulting construct is not a CORBA object. Specifically:

- 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 portably subclassed by user code, if it is represented as a class.

Note – The specific definition given for each piece of PIDL may override the general guidelines above. In such a case, the specific definition takes precedence.

3.2 Rules for Mapping Pseudo-objects

Unless otherwise indicated below, an OMG-defined pseudo-object defined by pseudo-IDL corresponds by default mapped to a Lisp class whose name is given by the scoped symbol corresponding to that pseudo-interface. Each pseudo-operation is mapped to a function whose name follows the corresponding rules for mapping of operations of interface.

3.2.1 Example

Considering the following pseudo-IDL:

```
module fum {
pseudo interface foo {
long bar (in string x);
};
};
```

The FOO pseudo-interface would correspond to a Lisp type named **fum:foo**. Evaluation of the form **(bar x "hello")** would return a value of type **corba:long** if **x** is of type **fum:foo**.

3.3 Certain Exceptions

The standard CORBA PIDL uses several exceptions: Bounds, BadKind, and Invalid-Name.

These are mapped as if they were standard user exceptions and inherit from **corba:userex-ception**.

The Bounds and BadKind exceptions map to conditions named **corba:type-code/bound**s and **corba:typecode/BadKind**. The Bounds exception is named **corba:Bounds**. This follows the usual mapping for the IDL defined in the CORBA core specification.

3.4 Environment

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

Since **conditions** in Lisp are first class objects, we define **Environment** simply as an **exception**:

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

3.5 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 to a class named **CORBA:NamedValue** via the following PIDL 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;
pseudo interface NamedValue{
attribute Identifier name;
attribute any argument;
attribute long len;
attribute Flags arg_modes;};
```

There is a corresponding constructor named **CORBA:NamedValue** that takes keyword initializers **name**, **argument**, and **Flags**. The default value of the **Flags** keyword initializer is **corba:ARG_IN**. The default value of the **name** keyword is the empty string **""**.For**OUT** parameter, portable applications shall set the **argument** attribute of a **NamedValue** to **nil** (i.e., the functionality in which an **OUT** argument parameter is set to point to a non-null storage pointer is not supported in this mapping).

3.6 NVList

An **NVList** is used in the DII to describe arguments and in the context routines to describe context values. An **NVList** is mapped to the type **CORBA:NVList**, which denotes the type of proper lists each of whose elements are of type **CORBA:NamedValue**. The standard list manipulation routines may be used to create such a list. The **create_list**, **get_count**, and **add_item** pseudo-operations are not mapped (this functionality is provided implicitly by Lisp list operations).

3.6.1 Example

```
(setq x
(list
(corba:NamedValue :name "test" :argument nil)
(corba:NamedValue :name "test2" :argument
(corba:any :any-value 4 :any-typecode corba:tc_long)
:flags corba:ARGS_INOUT))
(typep x 'corba:NamedValue)
----> T
```

3.7 Context

A **Context** is used in the DII to specify a **context** in which **context** strings shall 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 propname, in any propvalue);
void set_values (in NVList values);
void delete_values (in Identifier propname);
NVList get_values (in Identifier start_scope,
in Flags op_flags,
in Identifier pattern);
```

3.8 Request

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

```
pseudo interface Request {
readonly attribute Object target;
readonly attribute Identifier operation;
readonly attribute NVList arguments;
readonly attribute NamedValue result;
```

```
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_out_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();
```

The corresponding constructor is named **corba:request** and takes keyword initializers **target**, **operation**, **arguments**, and **ctx**. The value of the **ctx** attribute defaults to the current context.

3.8.1 Example

Suppose **(typep x ’corba:Request**).

Then the invocation:

```
(op:target x);
```

shall return the instance of **corba:Object** that is the target of the request, and **(operation x)** shall return the string that is the value of the **operation** attribute of **x**.

3.9 Dynamic Invocation Interface

3.9.1 Dynamic Invocation Interface Convenience Function

The following function is provided as a convenience interface to the DII.

The function **corba:funcall** with syntax:

corba:funcall *operation-designator target* &rest *params*

invokes the operation named by the *operation-designator* on the object denoted by the *ta get* parameter with parameters the *params*.

target should be an instance of *corba:Object*.

Operation-designator should be a symbol or a string.

An *operation-designator* denotes a particular declared IDL operation. If it is a string, it must be either the name of the operation or the fully scoped IDL name of the operation.

If it is a symbol in the *OP* package or the *keyword* package, it denotes the operation whose uppercased name is the print-name of that symbol. Otherwise, the *operation-designator* is interpreted as the "full scoping symbol" for the operation.The module in which the operation is declared corresponds to the name of the package of the symbol, and the name is of the form interface-name/operation-name, where interface-name is the uppercased-name of the interface (or abstract interface) in which the operation is declared and *operation-name* is the uppercased-name of the operation.

If target is not local and if *corba:funcall* is unable to determine the actual signature of the operation in sufficient detail to marshal the arguments, the condition CORBA:FUNCALL_MARSHAL shall be thrown. This exception shall inherit from CORBA:MARSHAL.

The values returned and the exceptions signalled by **corba:funcall** shall be consistent with the standard mapping for operation: if the operation completed successfully, the result and all out or inout parameters are returned in order of their declaration; otherwise, the exception signalled by the operation is returned.

3.9.2 Example

Consider the following IDL:

```
module outer {
module inner {
interface A {
exception tt {string x;}
void foo();
string fum (in long long bar) raises (tt);
short st(in char s, inout boolean y, out float z)
};
```

Suppose target is bound to an instance of **outer:inner/a**.

Then sample invocations might look like this:

```
(corba:invoke "foo" target)
--->[no values returned]

(corba:invoke "outer::inner::A::fum" target -31415926)
--->"PI"

(corba:invoke ’op:fum target -100)
---->[condition of type outer:inner/tt]

(corba:invoke ’outer/inner:a/st #\B nil)
---->134 T 1.34
```

3.10 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{
readonly attribute Identifier operation;
Context ctx();
void arguments(inout NVList nv);
void set_result (in any val);
void set_exception (in any ex);
```

3.10.1 Example

Suppose **x** is bound to an object of class **CORBA:ServerRequest**. Then the invocation **(op:operation x)** returns the string representing the operation corresponding to the request.

Additional detail on the use of **ServerRequest** is given in the “Server-Side” chapter of this document.

3.10.2 TypeCode

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

```
module CORBA{
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,
tk_value,tk_value_box,tk_native,tk_abstract_interface};

typedef short ValueModifier;
const ValueModifier VM_NONE=0;
const ValueModifier VM_CUSTOM=1;
const ValueModifier VM_ABSTRACT=2;
const ValueModifier VM_TRUNCATABLE=3;

typedef short Visibility;
const Visibility PRIVATE_MEMBER=0;
const Visibility PUBLIC_MEMBER=1;
};
pseudo interface TypeCode {
exception Bounds{};
exception BadKind{};
boolean equal (in TypeCode tc);
boolean equivalent (in TypeCode tc);
TypeCode get_compact_typecode();

TCKind kind();
RepositoryId id() raises (BadKind);
Identifier name() raises (BadKind);
//for struct, union, enum, value, value_box, and except
unsigned long member_count() raises (BadKind);
Identifier member_name(in unsigned long index) raises (BadKind, Bounds);

//for struct, union, value, value_box, 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);

//for fixed
unsigned short fixed_digits() raises (BadKind);
short fixed_Scale() raises (BadKind);

//for value
Visibility member_visibility (in unsigned long index) raises (BadKind,
Bounds);
ValueModifier type_modifier() raises (BadKind);
TypeCode concrete_base_type() raises (BadKind);
}; };
```

The **TypeCode** pseudointerface maps to the lisp class named **corba:TypeCode**. The operations defined on this class follow the pseudo-IDL above.

3.10.3 Example

Suppose **tc** is bound to a typecode representing a **struct** with the three members.

```
(op:member_count tc)
---> 3
CORBA:VM_CUSTOM
---->1
```

3.11 ORB

3.11.1 ORB initialization

The pseudo-IDL for ORB initialization is:

```
module CORBA {
typedef string ORBid;
typdef sequence<string> arg_list;
ORB ORB_init(inout arg_list argv, in ORBid orb_identifier);
};
```

The **ORB_init** pseudo-operation is mapped to a function named **CORBA:ORB_init**. Evaluation of **(corba:orb_init argv orb_identifier)** returns an instance of the **CORBA:ORB** class and a sequence of strings when **argv** is a sequence of strings and **orb_identifier** is a string. The semantics of the arguments **argv**, **orb_identifier**, and the returned values follow the definitions of these values in the CORBA specification.

Evaluation of **(corba:orb_init)** returns the ORB object that would be returned by invoking **(corba:orb_init nil "")** .

3.11.2 Example

```
(CORBA:ORB_init :orb_identifier "My ORB" :vendor-extension-key "Vendor-extension")

(CORBA:ORB_init)
```

3.11.3 ORB pseudo-object

The ORB pseudo-interface is mapped to a class named **CORBA:ORB**. The operations defined on the class follow the rules for mapping pseudo-IDL. The pseudo-IDL below is intended to follow the pseudo IDL given in the CORBA specification.

```
module CORBA{
exception PolicyError{PolicyErrorCode reason;};

typedef string RepositoryId;
typedef string Identifier;

typedef unsigned short ServiceType;
typedef unsigned long ServiceOption;
typedef unsigned long ServiceDetailType;

const ServiceType Security = 1;

struct ServiceDetail{
ServiceDetailType service_detail_type;
sequence<octet> service_detail;
};

struct ServiceInformation{
sequence<ServiceOption> service_options;
sequence<ServiceDetail> service_details;
};
pseudo interface ORB {
typedef string ObjectId;
typedef sequence <ObjectId> ObjectIdList;
```

```
exception InvalidName {};

string object_to_string(
in Object obj
);

Object string_to_object(
in string str;
);

//Dynamic Invocation related operations
void create_list(
in long count;
out NVList new_list
);

void create_operation_list(
in OperationDef oper,
out NVList new_list
);

void get_default_context(
out Context ctx
);

void send_multiple_requests_oneway(
in RequestSeq req
);
void send_multiple_requests_deferred(
in RequestSeq req
);

boolean poll_next_response();

void get_next_response(
out Request req
);

//Service information operations

boolean get_service_information (
in ServiceType service_type,
out ServiceInformation service_information
);

ObjectIdList list_initial_services();

//Initial reference operation

Object resolve_initial_references(
in ObjectId identifier
) raises (InvalidName);

//Type code creation operations

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
);
```

... [The other Typecode creation operations exactly follow the pseudo IDL given in the CORBA specification and are elided here].

```
//Thread related operations

boolean work_pending();
void perform_work();
void run();
void shutdown(in boolean wait_for_completion);
void destroy();

//Policy related operations
Policy create_policy(
in PolicyType type,
in any val) raises (PolicyError);

//Value factory operations
ValueFactory register_value_factory(
in RepositoryId id,
in ValueFactory factory
);

void unregister_value_factory(in RepositoryId id);
ValueFactory lookup_value_factory(in RepositoryId id);
```

3.11.4 Example

Suppose **(typep orb ’CORBA:ORB)** is **T**. Then the following invocations may be made:

(**work_pending** orb)

```
(run orb)
```

3.12 Object

The IDL **Object** type is mapped to the class **CORBA:Object**. It supports the operations defined in the pseudo-IDL below. The semantics of a pseudo-operation defined in the pseudo IDL below follow the semantics defined in the CORBA specification for the pseudo-operation whose name is the given pseudo-operation with the prepended underscore elided.

The **_is_nil** pseudo operation is mapped to the standard Common Lisp function **null**.

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

```
module CORBA{
interface DomainManager; //forward declaration
typedef sequence<DomainManager> DomainManagersList;
interface Policy //forward declaration
typedef unsigned long PolicyType;

interface Context; //forward declaration
typedef string Identifier;
interface Request; //forward declaration;
interface NVList; //forward declaration;
struct NamedValue{}; //an implicitly well-known type
typedef unsigned long Flags;
interface InterfaceDef; //forward declaration
enum SetOverrideType{SET_OVERRIDE, ADD_OVERRIDE};

pseudo interface Object{
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);
void _create_request(
in Context ctx,
in Identifier operation,
in NVList arg_list,
inout NamedValue result,
out Request request,
in Flags req_flags);
Policy _get_policy(in PolicyType policy_type);
DomainMangersList _get_domain_managers();
Object _set_policy_overrides(
in PolicyList policies,
in SetOverrideType set_add);
};
```

};

3.12.1 Examples

Suppose the variable x is bound to an instance of class CORBA:Object. Then the following are legal invocations: **(_is_equivalent x x) (_hash x 8777777)**.

3.12.2 Principal

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

3.13 DynAny

The **DynAny** data type is mapped to a class named **DynamicAny:DynAny** (equivalently, the class named **OMG.ORG/DynamicAny:DynAny**). The **DynAnyFactory** interface is mapped to a class named **DynamicAny:DynAnyFactory**. The definitions to these interfaces is given in IDL and follows the standard IDL mapping, except that the CORBA specification defines additional locality restrictions on their use. The usage of these classes thus follows the standard mapping.

3.13.1 Example

Suppose x is bound to an instance of **DynamicAny:DynAny**. The string **"foo"** may be inserted into **x** either via:

```
(insert_string x "foo")
```

or

```
(insert_any x "foo")
```

If **y** is bound to an instance of the class **DynamicAny:DynAnyFactory**, then a **DynamicAny** may be created via:

(**create_dyn_any x "foo"**)

3.14 The IDL Compiler

The function **CORBA:IDL** when applied to a single argument that is a pathname designator defines within the Lisp world in which it is invoked all data types, packages, classes, functions, and constants defined by the denoted IDL file. This may entail redefining classes or types.

Note – Pathname designator is defined in the ANS Specification (p. 26-35). Loosely speaking, it is a string that names a file, a "pathname object" that represents the name of that file, or a stream associated with that file.

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 value returned is an object of type **CORBA:Repository** and represents an Interface Repository representing the IDL file given as input. The returned object shall contain representation of the datatypes defined by the IDL file or shall be **nil.**

3.14.1 Example

Suppose the file named **"foo.idl"** contains:

```
module example {
struct y {long long zz;};
const long long c = 1000000000000000;
};
```

then the invocations

```
(CORBA:IDL "foo.idl")
(setq ex (example:y :zz example:c))
```

will bind the variable **ex** to an instance of **CORBA:struct** whose lone **zz** field has value equal to **1000000000000000**.

Server-Side 4

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

Contents

This chapter contains the following sections.

4.1 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**. Note that the full name of the PortableServer package is **OMG.ORG/PortableServer**, so that the types named here can also be specified as **OMG.ORG/PortableServer:Servant**, **OMG.ORG/PortableServer:ServantLocator/Cookie**.

The class **PortableServer:Servant** supports several operations designed for convenient interaction with the POA.

Application of the function named by **op:_this** to an instance of class **PortableServer:Servant** behaves as follows:

- Within the context of a request invocation on the target object represented by the servant, it allows the servant to obtain the object reference for the target CORBA object it is incarnating for that request.
- Outside the context of a request invocation on the target object represented by the servant, it allows a servant to be implicitly activated if its POA allows implicit activation. This requires the activating POA to have been created with the **IMPLICIT_ACTIVATION** policy. If the POA was not created with the **IMPLICIT_ACTIVATION** policy, the **PortableServer::WrongPolicy** exception is thrown. The POA used for implicit activation is gotten by invoking **op:_default_POA** on the servant.
- Outside the context of a request invocation on the target object represented by the servant, it will return the object reference for a servant that has already been activated, as long as the servant is not incarnating multiple CORBA objects. This requires the POA with which the servant was activated to have been created with the **UNIQUE_ID** and **RETAIN** policies. If the POA was created with the **MULTIPLE_ID** or **NON_RETAIN** policies, the **PortableServer::WrongPolicy** exception is signalled. the POA is generated by invoking **op:_default_POA** on the servant.

4.2 Dynamic Implementation

DSI servants shall inherit from the class **PortableServer:DynamicImplementation**, which in turn inherits from **PortableServer:Servant**. This class is defined via the following pseudo IDL:

```
module PortableServer{
pseudo interface DynamicImplementation(servant) {
void invoke (in ServerRequest request);
RepositoryId primary_interface(in ObjectId oid, in POA poa);
};
```

The class **PortableServer:DynamicImplementation** inherits the **op:_this** method from the **PortableServer:servant** class.

The **op:invoke** method, whose signature is specified in pseudo IDL above, receives requests issued to any CORBA object incarnated by the DSI servant and performs the processing necessary to execute the request.

The **op:primary_interface** method receives an **ObjectId** value and a **POA** as input parameters and returns a valid **RepositoryId** representing the most derived interface for that oid.

4.3 PortableServer Functions

Convenience functions are provided for conversion of **ObjectIds** to and from strings:

(PortableServer:Oid-to-string oid) (PortableServer:string-to-oid string)

These functions take respectively an **ObjectID** and a string and return the corresponding string or **ObjectID**.

4.4 Implementation objects

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

4.5 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 Feature 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 **portableServer:servant**.

Otherwise, if the interface has base interfaces named **A**, **B**, **C**, etc., 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**, etc.

4.5.1 Example

Consider the following IDL:

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

The corresponding Lisp hierarchy could look like this:

```
(defclass example:foo-servant(example:foo portableserver:servant)(..))
(defclass example:bar-servant(example:bar portableserver:servant)(..))
(defclass example:fum-servant (example:fum example:foo-servant example:bar-servant)(...))
```

The class diagram of the IDL is pictured in the section Mapping for Interface (see Section 2.8, "Mapping for Interface," on page 2-10). The class diagram for the generated Lisp is pictured below.

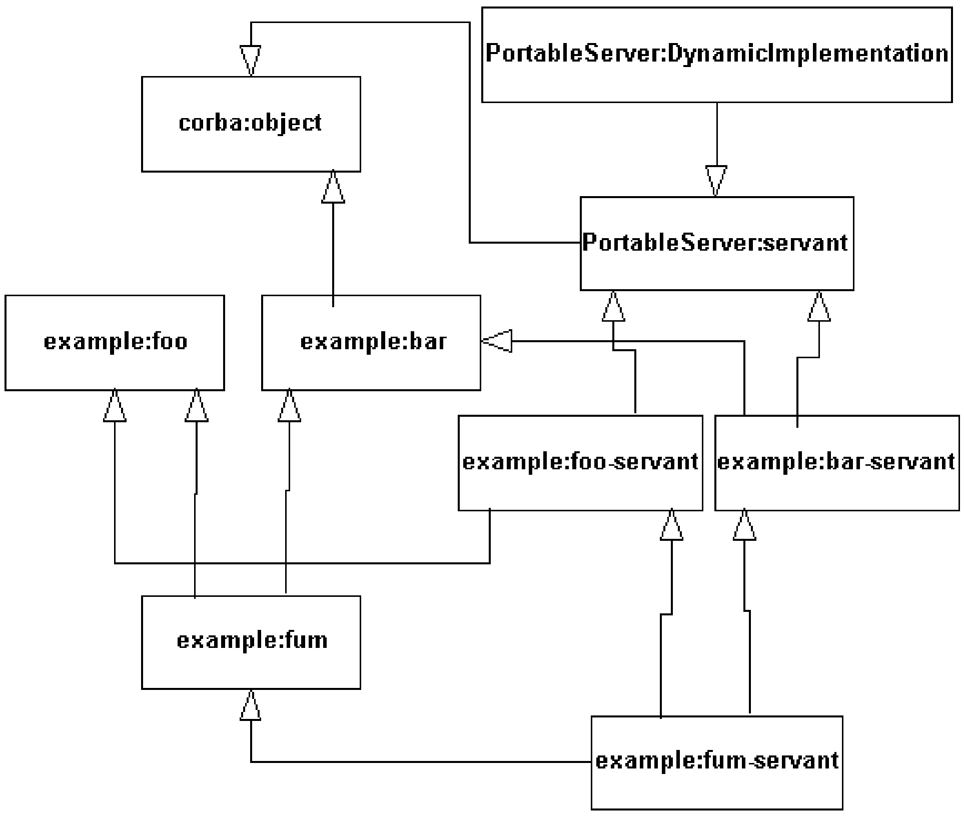

Figure 4-1 Lisp hierarchy corresponding to example 4.6.1

4.6 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.

4.6.1 Syntax of corba:define-method

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

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

```
receiver-specifer ::= (receiver-name receiver-class)
receiver-name ::= symbol
receiver-class ::= symbol
parameter-specifier ::= symbol
```

4.6.2 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 an attribute declared in an IDL interface implemented by the class named by the symbol **receiver-class**.

The number of **parameter-specifier**s listed in the **normal-lambda-list** shall 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 **setf**, the corresponding **operation-name** shall 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** shall 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.

It is not required that the invocations of **corba:define-method** that do not conform to the above syntax signal an error, although an implementation may so signal.

4.7 Examples

4.7.1 Example: A Named Grid

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

4.7.1.1 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);
}
```

4.7.1.2 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)())
```

4.7.1.3 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”)))
```

4.7.1.4 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 correspond to the **attribute** name that 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))
```

4.7.1.5 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 **operation**s 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”
```

Detailed Design Choices A

This chapter is a detailed description of the choices involved in selecting the mapping for each particular IDL element.

The design decisions in this chapter are subordinate to the general design principles discussed in Chapter 3.

The contents of this chapter are not normative and are not referred to by any normative section of this document.

A.1 Mapping for Feature

Several alternative mappings for the Feature meta-class were possible.

A.1.1 MM Naming Features

The question of the name to assign to the Lisp correspondent for an instance of the Feature metaclass proved to be the most difficult and controversial of the design decisions.

One obvious alternative would be to name Feature in the same way that Classifiers are named.

But Feature is explicitly not a subclass of Classifier, and for good reason: a Feature is only used in conjunction (indeed, in juxtaposition) with the Classifier that is its owner. Thus, explicit disambiguation of the Feature owner is not necessary in the current CORBA/UML object model. When and if this metamodel is extended to require such disambiguation, it would be entirely reasonable to provide addition functions for selecting the Feature from its name and the explicit name of its owner Classifier. Indeed, we have chosen to support this paradigm in one the DII invocation model.

In this way, our naming convention follows the Java mapping paradigm, rather than the C mapping paradigm.

Given that theoretical considerations perhaps could not entirely determine whether to require explicit naming of the Classifier of which a Feature is a feature, it is tempting to rely upon the wide body of praxis amongst two commercial ORB vendors, three ORBs, and many commercial users of these ORBs in disparate industries which has seemed to us to indicate a strong preference, albeit informally reported, among users for the Java-style Feature mapping. Nevertheless, this fact in itself is not persuasive due to the manifest influence of self-selection bias: those users who prefer the C-style Feature mapping may either have eschewed use of the commercial ORBs supporting the Java-style Feature mapping entirely, or they may have simply declined to report their opinions of the mapping. In any case, an ORB implementation is not proscribed from supporting C-style naming for Feature by this specification.

A.1.2 Feature Package

In any case, our decision not to require explicit naming of the Classifier that owns a Feature in using the Feature forces us to determine a particular package in which the symbol corresponding to the name of such a Feature shall reside. Long names are difficult to use; short names may conflict with existing usages. We chose a long name for disambiguation and a short nickname.

The minimal usage would have been to map to KEYWORD, as symbols in this package are simpler to use in certain ways than symbols in other packages when their symbol-value need not be set. Again, this usage could conflict with existing user code; because of CORBA’s common usage scenario of integration of legacy systems, we wanted to avoid any such legacy problems. Keywords are also not acceptable as slot names.

We also considered for some time various sophisticated tricks with the Lisp package import and package using functions. Our experience in practice is that the small space savings are rarely if ever worth the non-transparency of the resulting code.

We also considered mapping to keyword package together with prefixing the name of the Feature with a character such as "." or "/".

A.1.3 BehavioralFeature

The mapping of invocation of **BehavioralFeature** might seem to require overhead at invocation time due to the mismatch in lambda-lists between the declared IDL operation and the actual generic function named by that **BehavioralFeature**. Fortunately, compiler macro can solve this problem, if it does prove to be a problem, without affecting portability.

A.1.4 Feature visibility

The question of the mapping for **StructuralFeature** visibility arose only in valuetype mapping. We initially considered inhibiting accessor generation for private **StructuralFeatures**. We rejected this however and map visibility in the same way as we map the **isAbstract** attribute of **GeneralizableElement**: as a restriction on portable user code.

A.2 Names

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

A.2.1 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.

A.2.2 Nesting

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

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

A.2.3 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 shall be escaped for the default Lisp reader.

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

International characters

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

CORBA 2.3 eliminates this issue.

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.

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.

A.2.4 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.

A.2.5 Prefixes

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

An alternative is to use **lisp_package_prefix** in order not to conflict with future usages of this term by other languages. On the other hand, perhaps this usage will become standard and desirable.

A.3 Mapping of basic types

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

Each basic type is implicitly viewed as a classifier in the CORBA package, and is thus mapped to a Lisp type specifier in the CORBA package, following the standard mapping principles, regardless of whether an equivalent Lisp type already exists.

A.3.1 boolean

We considered mapping this type to the Lisp values defined by **generalized boolean**, which is easier to use in certain cases, or mapping to **boolean**, which may be simpler. The original implementation of this type was in fact to **boolean**, but it became clear that in practice the mapping to **generalized boolean** was simpler to use because so many standard Common Lisp predicates return generalized boolean.

A.3.2 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.

A.3.3 long double

ANSI Lisp does not require support for **long double**; and some Lisp vendors also do not support it on some platforms. However, **rational** can always be used as a stopgap, and our mapping is thus implementable on any Lisp.

A.4 Mapping for struct

The mapping for **struct** we chose is consistent with the standard mapping for **Classifier**. Each member of the struct is viewed as a **StructuralFeature** of the same Name with the standard naming convention and accessors for these.

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.

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.

A.5 Mapping for exception

From the point of view of UML, an exception class is viewed as a Classifier whose members correspond to **StructuralFeatures** of the same name.

User exception classes and system exception classes are considered to inherit (as Classifier elements) from the **UserException** and **SystemException** Class elements in the CORBA package.

The mapping of exception then follows directly from our standard mapping for Classifier.

The only question is the superclasses of **CORBA:Exception**, **CORBA:UserException** and **CORBA:SystemException** not determined by this mapping. Some amount of experience was needed to choose the current mapping.

A.5.1 condition hierarchy

Certainly **corba:exception**, the base class that arises from the UML mapping, must inherit from **condition**.

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 initially chose.

Experience with the way CORBA *system* exceptions are used in practice led us to the current mapping, in which system exceptions are mapped to subclasses of **error**. This allows macros like **ignore-error** to be used more easily with CORBA code. However, we retain the "old" mapping for *user* exceptions, as in practice some of these are clearly outside the specification for **error.**

It would certainly be a reasonable mapping for c**orba:exception** to inherit directly from **condition**. However, we think that exceptions should be signaled 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.

There is a theoretical backwards incompatibility problem with existing CORBA 2.2 ORBs whose handlers rely on **CORBA:SystemException** not inheriting from **error**. We would be extremely surprised if this problem affects any real code; and when it does, the problem is trivial to find and to correct.

A.5.2 Naming exception classes

We chose to name the classes corresponding to system and user **exception**s **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.

A.6 Mapping for enum

An enum type is considered as a Classifier; the mapping of the name follows the usual rule for Classier mapping.

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.

A.7 Mapping for union

A union is viewed as a **Classifier** with **StructuralFeatures** corresponding to each branch. Obviously since only one such branch is needed, only one slot need be physically represented. Otherwise our mapping follows our customary **Classifier** mapping.

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.

We considered the issue of automatic coercion of values to a union. We will consider this at a later version; there is not a pressing need for this convenience feature, and it has some semantic subtleties in the cases of ambiguous coercions.

A.8 Mapping for module

The mapping for module follows the standard UML mapping for the Classifier Package. All elements owned by the Package are named by symbols whose name is the name of that Package.

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.

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.

A.9 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 disjointed 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.

A.10 Mapping for sequence

We map IDL **sequence** to the Lisp type **sequence**.

There are several possible alternative mappings.

A.10.1 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.

A.10.2 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 permit 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.

A.10.3 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).

A.10.4 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 **list**s or **array**s 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 no artificial demarcation that the user shall remember between the mapping conventions.
- Our mapping is simpler and crisper in certain ways, according to some users.

A.10.5 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 slightly more complicated for the implementor of a method, since the method body shall be prepared to expect an arbitrary **sequence** (or a syntax in the method definition shall 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.

A.10.6 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.

A.11 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.

The integral typecode chosen in the case of integer operands is the smallest (from the CDR point of view) that it can hold. This is clearly consistent with well-known heuristics in Bayesian classification theory such as Minimum Description Length encoding.

A.12 Mapping for typedef

A typedef is considered a Classifier with no Features. The mapping follows the standard Classifier mapping rules.

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 at the top-level.

A.13 Mapping for interface

Our naming followed the standard naming conventions: an interface is viewed as a Classifier; attributes correspond to **StructuralFeatures**, operation to **BehavioralFeature**.

The interface mapping comprises several parts: Mapping from IDL for interface Class in UML model of IDL into mapped model (i.e., generation of auxiliary classes); Mapping of UML for interface into Lisp; implementation on server-side.

Mapping of the UML into Lisp is straightforward. Server-side mapping issues are discussed below.

A.13.1 Generation of auxiliary classes

We followed the Java mapping on generation of stub and skeleton classes corresponding to a given interface. We used as suffixes for these generated classes -proxy and -servant. It would be equally reasonable to use -stub and -skeleton; in any case this does not affect the semantics and we chose to be consistent with the large base of vendor code using the current prefixes.

Our mapping here avoided the aspect of the Java/C++ mappings in which certain user classes or IDL interfaces can nameclash with generated classes.

A.14 Mapping of valuetype

Valuetype is viewed in the standard way as a Classifier. Valuetype members are mapped to **StructuralFeatures**; operations are mapped to **Operation**.

Mapping valuetype must be done with great care because of interactions with POA, abstract, abstract interface, factory, GIOP, RMI Repository IDs, and custom marshal. Because the design goal of the Lisp mapping was to make the mapping as easy-to-use as possible, we require a lot of real-world use-cases to design the right mapping and make the right tradeoffs.

Therefore, our design decision was simply to retain consistency with our standard mapping principles and to defer to the extent consistent with minimal required portability the ease-of-use features to tool vendors. Although this design decision could theoretically have the consequence of engendering a proliferation of incompatible ease-of-use features (mostly macros), we felt the danger of this was outweighed in most cases by the danger of overspecifying features that turned out not to be widely used in practice but whose inclusion complicated exposition of the standard.

A.14.1 supports

The <<supports>> association from a Valuetype to its supported interfaces could have been implemented without inheritance, which could be more efficient in certain cases. We felt that mapping the <<supports>> association to inheritance was simpler.

Note that this diverges from the C++ 2.3 design decision which does not map **supports** into inheritance. The Java 2.3 mapping does map <<supports>> to inheritance, but onto the specific Operations interface associated with an interface, which for the Lisp mapping is implicit in the interface. We felt our mapping was possibly simpler to explain and we felt that at this time there was insufficient usage data to determine whether the concerns in the C++ supports mapping would in fact obtain in Lisp real-world usage scenarios, with its inherently looser typing.

A.14.2 unmarshalling and custom unmarshalling

We followed the C++ unmarshalling semantics, rather than the Java ones. The main difference here is that the C++ does not explicitly address unmarshalling of RMI repository ID's.

For custom classes we do choose to have generated classes inherit from **CustomMarshal**.

A.15 abstract valuetype

Abstract is not supported directly by Lisp, so we essentially only support this as a constraint on user code.

A.15.1 abstract interface

An abstract interface is mapped according to the normal Classifier mapping.

The key question in abstract interface was simply whether to inherit from Object. Our original mapping left this up to tool vendors, but for specificity we now require it. Because of our of our mapping for <<supports>>, it is the case that any usage of abstract interface must inherit from Object (indirectly).

A.15.2 Lisp-to-IDL

It is natural to assess the impact of our mapping on future Lisp-to-IDL mappings. We determined not to assess this question in the current mapping in order to simplify the job of creating our OBV mapping. As additional experience is gained this question can be readdressed.

A.15.3 Value box

We chose our mapping based on the fact that the non-primitives are passed by reference anyway in Lisp and can be checked for eq-ness by the compiler. Arguably we could have done the same for primitive types, but we decided it was simpler to use the C++-style mapping for these boxes.

A.16 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, we the current mapping is much simpler.

It would be desirable at some future time to allow URL’s as pathname designators. However, this is quite complex to describe (i.e., #include must be described, and the exact space of accepted URLs must be described) so we decided not to do so.

A.17 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:

Our pseudo-interface mapping is quite straightforward.

Our **is_nil** mapping was chosen to assure that pseudo-operations were not invoked on **nil**, which could theoretically cause some future problems with future **op:** generic function implementation.

In the case of the DII, we added several convenience features so that users do not have to specify the typecodes and return types of the calls.

Our original mapping followed the Java mapping, but we felt the ORB can get the typecodes just as well as the user.

In practice, many non-Lisp ORBs do not support **_interface**, so the exception signalled when the ORB is unable to infer the typecodes in using the DII is significant. We chose a subclass of **CORBA::MARSHAL**; an alternative would have been a new top-level CORBA system exception. However, since this invocation is local, we saw no need not to inherit from the existing exceptions.

We considered **_invoke** as a pseudo-operation on **Object**, and a top-level **corba:invoke**.We also considered putting the target first in **corba:funcall**.

For simplicity we followed the familiar "_" prefix in **Object** pseudo-operations. Although more elegant solutions were certainly possible, they were deemed not necessary in this case.

A.18 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.

We also considered a particular define-class macro, analogous to the define-method macro.

Since specification of the metaclass of implementation classes is unnecessary in most cases, we could simply follow the standard defclass syntax, replacing the metaclass specifiers by method specifiers. Attribute specifiers are mapped using expected syntax.

An example might clarify. Consider

```
module ex {
interface foo {
string foo ();
long fum(in long arg);
};};
```

A user implementation of this class could be defined via:

```
(corba:define-method my-foo (ex:foo-servant)
((my-slot :accessor my-slot))
(method foo () ("hello from method foo"))
(method fum (arg) (+ (my-slot this) arg))
(method foo :before (format t "Calling method foo
now...~%")))
```

Note that "this" is bound to the target in execution of each method body.

A similar compatible syntax is used for attribute specifiers. We eventually rejected this proposal for inclusion in this version of the mapping. In any case, it is easy for the user to provide such a macro if desired.

Lisp Concepts B

This chapter presents a simplified overview of some key Lisp concepts used in this mapping document. Since the ANS Lisp standard, on which this document is based is approximately one thousand pages long, it seemed useful to limn some of the key concepts.

Note – This appendix, of course, is non-normative, and in some cases *is* oversimplified or inaccurate. This appendix is not intended to serve as a Lisp reference. The ANS specification should be referred to for normative definitions.

B.1 Lisp evaluation

The life of a Lisp form typically comprises three phases:

1. read: A character representation of the form is read by the invocation of the Lisp function read on that string. This returns some Lisp value.
2. eval: This Lisp value is then evaluated, producing zero or more Lisp value or values.
3. print: These values are printed

Of course, the actual work is done in the evaluation phase. In examples we typically show the input and output of the forms to a simple Lisp listener that indeed simply reads, evaluates, and prints. In these examples in this specification, we preface the output of the listener with string "--->".

The evaluation of a form (a value) is simple enough. If the form is a list, the first element should be a function or the name of the function; that function is applied to the arguments that result when the remaining elements of the list are evaluated. Otherwise, the form itself is returned.

B.1.1 Example

(+2 3 (*1100000))

---> 100005

Here, the user has input the string "(+2 3 (*1100000))" to the listener, which has printed 100005. The result of evaluating the list (* 1 100000) is 100000, and the result of evaluating (+ 2 3 100000) is 100005.

This is often informally referred to as "The result of evaluating the form (+2 3 (*1100000))) is the number 100005", or "Applying the function named by the symbol + to the arguments 2, 5 and the result of applying the function * to the aguments 1 and 100000 is 100005.

B.2 Lisp values

A Lisp entity can be, among others, one of the following: *number*, *cons*, *symbol*, *object*, *type*, *class*, *function*.

A *number* is either an (arbitrary size) *integer*, an arbitrary-size *rational*, a floating point *number*, or a complex *number*.

B.3 Cons

A *cons* is basically a structure with two fields, the *car* and the *cdr*, each of which can hold any other Lisp object. A cons is notated by writing the ’**(**’ character, the representation for the **car**, the "**.**" character, the representation for the **cdr**, and the ’**)**’ character. If the **cdr** is itself a **cons** then the "**.**" and the parentheses that would normally surround that **cons** are elided. If the **cdr** has the value **nil** then the ’**.**’ character is omitted and that **cdr** is not printed.

A **list** is either a **cons** or **nil**.

Lisp has numerous built-in functions for manipulating lists. Some of the most common: **car** returns the **car** of a **list**, **cdr** returns the **cdr**, **list** constructs a **list** from its arguments, **append** splices together **list**s, **mapcar** maps a function over a **list**, and so on.

B.3.1 Example

(2 .3)

denotes the **cons** whose **car** is **2** and whose **cdr** is **3**.

(2 3 4) denotes the **list** with three elements, **2**, **3**, and **4**; equivalently, it is the **cons** whose **car** is **2** and whose **cdr** is the **cons** whose **car** is **3** and whose **cdr** is the **cons** whose **car** is **4** and whose **cdr** is **nil**: **(2 .(3 .(4 .nil)))** .

B.4 Arrays

Arrays are created with **make-array** and accessed with the **aref** function.

(setq a (make-array 5)) ; creates a 5-element 1-dimensional array

(setf (aref a 3) "hello") ; Sets the third element of **a** to the string **"hello"**.

(aref a 3) ; get the third element of the array

---> **"hello"**

Multidimensional arrays are created using calls like **(make-array '(2 3)**) for a 2 by 3 array and elements are accessed via **(aref a 1 2)** for the element at index **(1 2)** (all indices are 0-based).

A 1-dimensional array is called a **vector**.A **string** is a **vector** of **character**.

Various arguments to **make-array** can constrain the type of each element, change the base of the indices, align the array with another array, or allow the array to be extensible.

B.5 Types

A **type** is a set of objects. A type specifier is the name of the type. For example **integer** names the type of all integers, while the list **(integer 0 43)** names the type of integers from **0** to **43**.

If **p** is a type specifier then **(typep x p)** is **T** if and only if **x** is a member of the type denoted by **p**.

The type **generalized boolean** denotes all Lisp values, in which **nil** signifies false and non-nil **values** denote truth. Many Lisp built-in predicates return generalized booleans.

B.6 Symbols

A symbol has a **package** and a name. The package is essentially a named collection of symbols, and the name is a string. Packages have a **package-name** and any number of nicknames; any of these are said to name the package. A symbol is notated via **<package>:<name>** where **<package>** is a name of the package and **<symbol>** is the name of the symbol. The package specifier is omitted if it is understood.

A symbol can be external or internal in a package. In the latter case, two colons, not one, must be used in notating it. It is possible, although it can be complicated, to specify in certain lexical contexts that a symbol can be accessed without its package designator. All the symbols used in this mapping are external.

Symbols in the keyword package, also called keywords, are particularly convenient to use because they always have themselves as values and they can be notated simply via strings like "**:foo**" for the symbol named "**foo**" in the keyword package. Keywords are typically used to specify optional named parameters to functions.

A symbol has a value (which can be any object) and a function value. It can also name a type or a class; these namespaces are disjoint.

When a symbol is evaluated, its value is returned. The value of a symbol is set using the **setq** macro.

```
(setq a -1)
---> -1
a
---> -1
(defun a (x y) (+ x y))
a
-1
(a 3 1)
---> 4
```

B.7 Functions

A function is applied to its arguments:

(+2 3 4)

---> 9

Here + is not a function. Rather, + is shorthand for the symbol "**common-lisp:+**" whose function-value is the addition function.

The actual addition function can be obtained from its name via:

(function '+)

A shorthand for this is

#'+

Functions can be defined by the defun macro.

```
(defun my-splice (a b)
(cons a b))
(my-splice 2 3)
(2 . 3)
(my-splice "foo" '(x y z))
("foo" x y z)
```

The **defun** macro takes certain standard Lisp syntactic markers to specify the form of the argument list: whether keywords are used and if so which ones and the default value of their corresponding arguments; whether optional arguments are used; whether an arbitrary number of arguments may be passed.

B.8 Example

(defun keyword-example (x &key y z (foo "hello")))

```
(list x y z foo))
(keyword-example 3)
---> (3 nil nil "hello")

(keyword-example 1 :y 6 :z '(1 1) :foo "goodbye")
---> (1 6 (1 1) "goodbye")
(keyword-example 1 :z '(1 1) :foo "goodbye" :y 6)
---> (1 6 (1 1) "goodbye")
```

Functions are first class objects. They can be explicitly applied to lists of arguments in various ways. Functions may be redefined. Functions may be, and usually are, compiled.

B.9 Classes

A **class** object controls the behavior of its instances. The **metaclass** of an instance is the class of its class. Lisp defines various built-in meta-classes, but most user classes are instances of the standard metaclass standard-class. Classes multiply inherit and have slots that hold state.

```
(defclass furniture ()
(price :initform 0 :accessor get-price :initarg :price)
(id))
(defclass wooden ()
(kind :initform :oak :accessor get-kind :initarg :kind))
(defclass wooden-table (furniture wooden)
(legs :accessor get-legs :initarg :legs))
```

This defines three classes, a base-class **furniture** which inherits from **standard-object**, a class **wooden** that has a single slot holding the type of **wood**, and a subclass **wooden-table** that inherits from **furniture** and **wooden**.

An oak table with 3 legs and costing 10 dollars can be created via

```
(make-instance 'wooden-table :price "10 dollars" :kind :oak :legs 3)
```

The **class-of** function returns the **class** of its argument. The **class** of a value can be changed by the **change-class** function.

The definition of a class can be changed. The slots in its instances have the natural default behavior when the new class has a different set of slots, but the behavior of an instance whose class has changed definition can be modifed by specializing the generic function **update-instance- for-redefined-class**.

Similarly, if the class of an instance is changed, its behavior can be customized by using the function **update-instance-for-different-class**."

B.10 Methods

Methods are similar to functions except that they may dispatch on zero or more of their parameters. Methods belong to a generic function, which determines, when it is applied to arguments, the correct method to call.

For example, here is a method to print an invoice for a wooden table:

```
(defmethod print-invoice ((this wooden-table) &key note)
(print "Congratulations on purchasing this beautiful wooden table")
(print the price is: (get-price this))
(if note (print note)))
```

Now if x is an instance of a wooden table, an invoice is printed via

```
(print-invoice x)
```

or

```
 (print-invoice x :note "Net due in 90 days")
```

B.10.1 Auxiliary methods

A method may have associated **auxiliary** methods. Common types of auxiliary methods are **before** methods, which are always executed before the primary method is run; **after** methods, which are evaluated after the method is run; and **around** methods, which conditionally control the behavior of the primary behavior (an around method can call its associated primary method by using the **call-next-method** function.

For example, we can define an **around** method on **print-invoice** via:

```
(defmethod print-invoice :around ((this wooden-table) &key note)
(if
(equal note "Reserved")
(print "Warning: invoice should not be printed")
(call-next-method)))
```

If the passed in **note** keyword is the string **"Reserved"**, then a warning is printed and the invoice is not printed. Otherwise, the primary method is invoked as usual.

Around methods are common in distributed systems to obtain the functionality of smart proxies.

B.11 Macros

There are three types of macros in Lisp: reader macros, macros, and compiler macros.

A reader macro is used to lexically rewrite code before it has left the reader. For example, reader macros are used to give some characters special significance.

Normal macros transform forms before they are evaluated. They can perform arbitrarily complex manipulations on forms.

Compiler macros are only evaluated when a form is compiled. They are most often used for optimization reasons.

Macros are commonly used in Lisp programs to tailor the application language to the domain. For example, **corba:define-method** is a (very simple) macro.

B.12 Compilation

In examples Lisp forms are usually interpreted, but in real life they are normally compiled. Typically this produces an intermediate form, and optionally a file, that can be loaded and executed more quickly than the original form. The compiler is accessed via the functions compile and compile-file.

A form can be evaluated only when loaded time, when evaluated, or when compiled time. The **eval-when** construct is used for this.

```
(eval-when (compile) (form-to-be-executed-only-during-compile))
```

A form can also be executed at read time by prefacing it with the string **"#."**

B.13 Conditions

Lisp has a multiply inherited condition system. Conditions can be signalled with the signal function. They can be caught via the handler-case form:

```
(handler-case
(stuff-to-evaluate)
(error-type-1 (condition-signalled) (stuff-to-do condition-signalled))
((error-type-2) (condition-signalled (stuff-to-do condition-signalled))
```

Lisp also supports facilities for recovering from signalled conditions. Although this is important in using CORBA, as it is often convenient to provide the developer the option to reinvoke server-side operations if they've signalled an unexpected error, it is in practice unusual for the small-scale applications builder to need to implement systems that do this, and we will not discuss this here.

Index

Index