ASN.1

Communication between Heterogeneous Systems

Olivier Dubuisson

translated from French by Philippe Fouquart

http://asn1.elibel.tm.fr/en/book/
http://www.oss.com/asn1/booksintro.html

June 5, 2000

ASN.1
Communication between heterogeneous systems

by Olivier Dubuisson

ASN.1 (Abstract Syntax Notation One) is an international standard which aims at specifying of data used in telecommunication protocols. It is a computing language that is both powerful and complex: it was designed for modeling efficiently communications between heterogeneous systems.

ASN.1 was in great need of a reference book, didactic as well as precise and Olivier Dubuisson's book meets these demands. The language is comprehensively described from its basic constructions to the latest additions to the notation. The description of each of these constructions is wholly accessible and accurate. Many case studies of real-world applications illustrate this presentation. The text also replaces the language in its historical background and describes the context in which it is used, both from the application viewpoint and from that of other specification standards which use or refer to ASN.1.

This book is written by an expert of ASN.1, of its syntax and semantics, and clearly constitutes a reference on the language. It is intended for those merely interested in finding a complete and reliable description of the language and for programmers or experts who may want to look up for the proper usage of some constructions. The tools available on the website associated with this book will prove useful to both the proficient and the beginner ASN.1 user.

Michel Mauny
Project leader at INRIA, the French National Institute for Research in Computer Science and Control

Olivier Dubuisson *is a research engineer at France Télécom R&D, the Research & Development centre of France Télécom (formerly known as Cnet), where he is in charge of the ASN.1 expertise. He takes part in the language evolution at the ISO and ITU-T working groups. He has also developed various editing and analysis tools for ASN.1 specifications and assists the ASN.1 users at France Télécom in numerous application domains.*

Philippe Fouquart *graduated from Aston University, UK with an MSc in Computer Science and Applied Maths in 1997. He worked for Cnet on ASN.1:1994 grammar and later joined France Télécom R&D in 1999 where he used ASN.1 for Intelligent Network and SS7 protocols. He is now working on Fixed-Mobile Converged architectures and IP mobility.*

ISBN:0-12-6333361-0

To my parents

"All right, but do you think, notwithstanding so major an unknown factor, that you can unlock what signal it holds for us?" [...]

"Actually, I would count on a trio of distinct strata of classifications.

"First, you and I look at it casually and think of it as just confusing poppycock, foolish mumbo jumbo - noticing, though, that, as a signal, it's obviously not random or chaotic, that it's an affirmation of sorts, a product of a codifying authority, submitting to a public that's willing to admit it. It's a social tool assuring communication, promulgating it without any violation, according it its canon, its law, its rights.

"Who knows what it is? A bylaw? A Koran? A court summons? A bailiff's logbook? A contract for purchasing land? An invitation to a birthday party? A poll tax form? A work of fiction? A crucial fact is that, my work advancing, what I'll find rising in priority isn't its initial point of application but its ongoing articulation for, if you think of it, communication (I might almost say 'communion') is ubiquitous, a signal coursing from this individual to that, from so-and-so to such-and-such, a two way traffic in an idiom of transitivity or narrativity, fiction or imagination, affabulation or approbation, saga or song."

Georges Perec, *The Void* (translated by Gilbert Adair).

Contents

List of Figures

List of Tables

Foreword

Abstract Syntax Notation One (ASN.1) is a notation that is used in describing messages to be exchanged between communicating application programs. It provides a high level description of messages that frees protocol designers from having to focus on the bits and bytes layout of messages. Initially used to describe email messages within the Open Systems Interconnection protocols, ASN.1 has since been adopted for use by a wide range of other applications, such as in network management, secure email, cellular telephony, air traffic control, and voice and video over the Internet.

Closely associated with ASN.1 are sets of standardized encoding rules that describe the bits and bytes layout of messages as they are in transit between communicating application programs. Neither ASN.1 nor its encoding rules are tied to any particular computer architecture, operating system, language or application program structure, and are used in a range of programming languages, including *Java*, *C++*, *C* or *COBOL*.

The formal standards documents on ASN.1 and its encoding rules are published by the International Telecommunications Union-Telecommunications Sector (ITU-T), by the International Organization for Standardization (ISO) and by the International Electrotechnical Commission (IEC). Though the standards are very thorough and precise in their definitions, they are not easy to read.

The purpose of this book is to explain ASN.1 and its encoding rules in easy to understand terms. It addresses the subject at both an introductory level that is suitable for beginners, and at a more detailed level that is meant for those who seek a deeper understanding of ASN.1 and the encoding rules. Application protocol designers who need a solid understanding of ASN.1, and computer programmers who desire a clear and full understanding of the standardized encoding rules of ASN.1, will benefit from reading this book.

Olivier Dubuisson has done a very good job of describing ASN.1 and its encoding rules in this book. Starting with his overview of ASN.1, to his detailing of ASN.1 and its encoding rules, and finishing with his description of ways in which ASN.1 is today used in industry, he provides clear examples to help the reader better grasp what is being said. The material is presented in a form that will prove enlightening for technical managers who seek a general understanding of ASN.1, standard writers who wish to know ASN.1 in detail but without having to concentrate much on the encoding rules, as well as those implementers who may need to know the details of both ASN.1 and its encoding rules.

Olivier is a member of the ISO/IEC ASN.1 committee and is known for his deep theoretical and practical understanding of ASN.1 and its encoding rules. His focus on clarity of both ASN.1 concepts and the text used to describe them has proved very valuable.

Olivier has certainly accomplished his objective of providing a thorough yet clear description of ASN.1 and its encoding rules. The pages that follow contain the ins and outs of the ASN.1 standard and the encoding rules, and you will find that they are presented in an easy-to-understand and friendly manner. Readers will no doubt be delighted at having their desire for a fuller understanding of ASN.1 fulfilled and their questions answered by the material on the following pages.

Bancroft Scott
President, OSS Nokalva
Editor, ITU-T | ISO/IEC Joint ASN.1
and Encoding Rules Standards
Somerset, New Jersey, 23 Feb. 1999

Preface

Par excellence, ASN.1 (Abstract Syntax Notation One) is a formal notation that allows specifications of information handled by high level telecom protocols with no loss of generality, regardless of software or hardware systems.

Since its first standardization in 1984, ASN.1 has widened its scope out of Open System Interconnection (OSI) and benefited from numerous improvements, particularly in its 1994 release in which substantial functionalities related to telecommunication technological changes (high rate data transfer, multimedia environment, alphabets of developing countries, service protocol frequent updating, etc) were added.

In addition, more and more computing tools are available to make ASN.1 easier to handle. Finally, several standardized sets of encoding rules can be used for describing how these potentially complex data should be transmitted as bit or byte streams while keeping the way they are transparent to the specifier.

For those who deal with data transfer, whether simple or very complex, ASN.1 is a pre-eminent notation. However, little literature and no book encompass ASN.1 as a whole: from its semantics to its encapsulation in other languages (SDL, TTCN or GDMO) including encoding and its related tools.

Having taught ASN.1 in industrial courses at France Télécom R&D (the research & development centre of France Télécom, which was named "Cnet" at that time) and done my best to answer numerous questions from my insatiable colleagues on the subject for some years, convinced me that there was indeed material for a book. I have tried to consider several levels for reading, from the neophyte[1] without any concept of data transfer up to the expert. These reading directions are given in Figure 1 on page xxi, but the reader should not feel restricted in any

[1]The neophyte can discard the numerous footnotes of this book in the first reading.

way by them, and no doubt the specifier who is already familiar with ASN.1 and simply wants to look up a specific notion will go directly to the corresponding section: this book includes numerous cross references in order to guide readers more quickly to related notions and to help them build up their own way of reading.

For merely technical books are hardly ever entertaining, I endeavoured a few digressions that, hopefully, a rigorous reader will not hold against me!

Contents

This book is divided into four parts. The first is called 'Introduction and History of the Notation'. It starts with a metaphore about the 'amazing story of telecommunications' and proceeds with an introduction to general principles of data transfer and ASN.1 benefits. This chapter is dedicated to the reader who is not acquainted with this area.

Chapter 3 places ASN.1 in the historical context of OSI, and more specifically how it relates to the layers 6 and 7 (Presentation and Application). By means of a tutorial, Chapter 4 presents a real-world data transfer specification of a mail order company. Chapter 5 introduces the main concepts of ASN.1 along with examples (types, abstract values, modules, information object classes and information objects) to lead more smoothly to Chapter 6, which traces the outline of ASN.1 historical background and describes the different standardized versions. Finally, Chapter 7 presents various application domains of ASN.1.

Readers who would simply be looking for an overview of ASN.1 and its main advantages may, in the first place at least, focus on this first part, which is independent from the rest of the book.

The second part, titled 'User's Guide and Reference Manual', makes up the core of the book, since it describes full ASN.1 syntax and semantics. Chapter 8 provides an introduction to the reference manual, which contains nine chapters. Those are arranged by increasing difficulty level. Unfortunately from a pedagogical viewpoint, it is impossible to separate ASN.1 into fully independent sections. The sub-sections called 'User's Guide' are aimed at the beginners who may leave the 'Reference Manual' to subsequent readings.

The third part, 'Encoding Rules And Transfer Syntaxes', includes three chapters that describe the four standardized transfer syntaxes associated with ASN.1. The following chapter focuses on other sets of

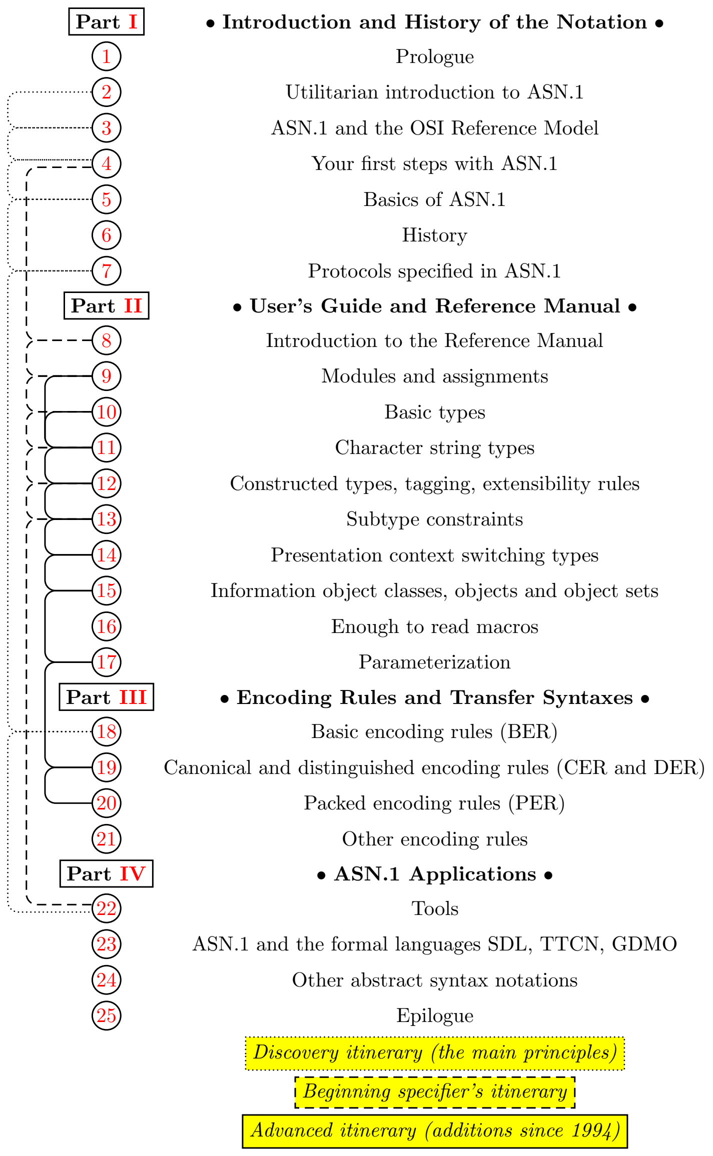

Part I
• Introduction and History of the Notation •
1 Prologue
2 Utilitarian introduction to ASN.1
3 ASN.1 and the OSI Reference Model
4 Your first steps with ASN.1
5 Basics of ASN.1
6 History
7 Protocols specified in ASN.1
Part II
• User's Guide and Reference Manual •
8 Introduction to the Reference Manual
9 Modules and assignments
10 Basic types
11 Character string types
12 Constructed types, tagging, extensibility rules
13 Subtype constraints
14 Presentation context switching types
15 Information object classes, objects and object sets
16 Enough to read macros
17 Parameterization
Part III
• Encoding Rules and Transfer Syntaxes •
18 Basic encoding rules (BER)
19 Canonical and distinguished encoding rules (CER and DER)
20 Packed encoding rules (PER)
21 Other encoding rules
Part IV
• ASN.1 Applications •
22 Tools
23 ASN.1 and the formal languages SDL, TTCN, GDMO
24 Other abstract syntax notations
25 Epilogue
Discovery itinerary (the main principles)
Beginning specifier's itinerary
Advanced itinerary (additions since 1994)

Figure 1: Reading directions

encoding rules that may be applied to an ASN.1 abstract syntax. This part can be set aside by specifiers, who need not, in general, be concerned with the way data described in ASN.1 are transmitted.

The fourth part takes a more applied perspective. Chapter 22 shows what is meant by 'ASN.1 compiler' and tackles the problems that are come up against when dealing with the automatic treatments of specifications. Chapter 23 introduces the use of ASN.1 in the context of other formal languages, such as SDL for formalizing telecommunication protocols, TTCN for testing these protocols and GDMO for network management. Finally, Chapter 24 presents other abstract syntax notations and compares them with ASN.1.

All the abbreviations used in this book are summarized from page 515 onwards. An index of the various concepts, keywords and names of the grammar elements is on page 543.

Considering the number of chapters and the potential variety of readers, we suggest several guidelines for reading this text in Figure 1 on page xxi.

Contacts

On the Web, discussions about ASN.1 can be found on the asn1@oss.com mailing-list. Unrelated to the compiler developed by OSS Nokalva[2], this list is very reactive and of a high technical level; it gathers all the international experts on ASN.1 and reaches around 250 subscribers. To subscribe, send an e-mail to asn1-request@oss.com with '`subscribe asn1` *`your-name`*' in the body of the message.

Readers who may wish to contact me or ask me questions can send e-mail to asn1@rd.francetelecom.fr or surf to the web site http://asn1.elibel.tm.fr associated with this book.

This text is very unlikely to be error-free and the ASN.1 standard is not frozen. All comments are welcome (please note the section, page and line number in fault); they will be collected on a special web page[3] on our site and I shall maintain a mailing list[4] of the readers who want to be informed of the corrections together with the latest news of the site. Finally, for the careful readers who would wish further detail, the text provides numerous URLs. As their life may happen to be dreadfully

[2]http://www.oss.com
[3]http://asn1.elibel.tm.fr/en/book/errata/
[4]http://asn1.elibel.tm.fr/en/news/

and unfortunately short as everyone knows, these will be updated on our web site[5] too.

Happy reading!

Acknowledgements

I am very much indebted to Bancroft Scott, ISO and ITU-T editor of the ASN.1 standards, for the technical quality of this book. Bancroft has always answered precisely and promptly the numerous questions I ask him (note the tense) almost daily. He strongly supported the project, and I thank him warmly for that. He is somehow one of the two living memories of ASN.1.

John Larmouth, ISO rapporteur for ASN.1, is its second memory, something everybody should be convinced of when reading his book [Lar99]. His historical viewpoints on aspects too old for me (no offense meant!) have been of great help. John is a first-rate contributor on the standard and I hope our readers will find complementarity in our respective books.

A few years ago, when he was still a student and about to move to England for his studies, Philippe Fouquart worked with me on ASN.1:1994 grammar. It has been a pleasant surprise to see him back, now as a colleague at France Télécom. He promptly agreed to translate this book into English and I'm content to see how precisely, but also how easily, he managed to put across the very ideas which I meant to emphasize. His suggestions also provided improvements to the original text (which, hopefully, the French-speaking readers will benefit from in another edition). I must pay tribute to him for the remarkable quality of his work.Philippe has also seen to compiling the original LaTeX source files into a PDF that could claim the name, fixing up fonts, cross-references and index bugs: thanks for cutting the Gordian knot!

Michel Mauny is a friend I have been able to count on since my early struggles with ASN.1. His continuous support, his thoughtful advice, his open-mindness and his serene attitude have been of invaluable help to me. It has been my pleasure to work with him.

I gratefully aknowledge the help and support of Rodolphe Pueyo. I would like to thank him for providing me with thoughtful criticism and advice while reviewing this book. I'm proud of supervising Rodolphe's PhD thesis on ASN.1.

[5]http://asn1.elibel.tm.fr/en/book/links/

The web site[6] associated with this book is dedicated to Philippe Fouquart, Frédéric Duwez, Guillaume Latu, Pierre-Marie Hétault, Yoann Regardin, Stéphane Levant and Guillaume Moraine, who throughout their placements at France Télécom R&D, always worked enthusiastically, with energy and proficiency on the development of original tools, some of which are available on this site.

These tools would not have seen the light without Christian Rinderknecht's trail-blazing work on ASN.1 parsing. I have always appreciated his Cartesian approach of ASN.1 grammar and semantics. His theoretical and applied results are the foundation of our developments.

This book could not have been written without the support of Roland Groz and Pierre-Noël Favennec who encouraged me in giving concrete expression to my project.

I am grateful to Anne-Marie Bustos's contribution, with whom I had the pleasure to give courses on ASN.1 at France Télécom R&D. The 'first steps' in Chapter 4 are very much inspired by one of the tutorials she gave.

I am particularly grateful to a number of people for carefully reviewing draft material of this book: Bruno Chatras, Jeannie Delisser, Fabrice Dubois, John Ehasz, Roland Groz, Pierre-Marie Hétault, Jean-Paul Lemaire (who also provided some examples and good advice), Sergey Markarenko, Guy Martin, Rodolphe Pueyo, Yoann Regardin, Christian Rinderknecht, Bancroft Scott, Conrad Sigona, Paul Thorpe and Daniel Vincent.

I should like to also extend my thanks to the librarian assistants at France Télécom R&D and to Arnaud Diquelou at AFNOR for their invaluable help during my biographical roving.

Olivier Dubuisson
Perros-Guirec, France
February 1999

Translator's notes

Though an absolute beginner in the area, there was not much hesitation to go in for Olivier's offer of translating his book from French into English, even if tackling quotes from Moliere's or Flaubert's seemed a bit daunting for an amateur at first. For this, I ought to apologize in

[6] http://asn1.elibel.tm.fr/

advance to the well-read purists who would be offended by such careless attention... Generally speaking, because this translation is not specifically aimed at native English speakers, we have tried to keep the phrasing as simple as possible in so far as talking about ASN.1 can be made simple! The American spelling, typography and syntax, which appear throughout the text, have been used for editorial purposes.

This translation is meant to be consistent with the standard documents. Whenever, for clarity's sake, the vocabulary has to differ from the standard terms, the correspondence is annoted, generally by means of footnotes. To contribute to the added value of this book to the standard documents, we deliberately used this variety to put across the concepts rather than the vocabulary employed to describe them since focusing on words rather than ideas is sometimes the weak spot of the standard texts. We found that such a variety would be of valuable help as long as it would remain unambiguous: rephrasing proves to be a good way to learn. In the Reference Manual, which is more closely related to the standard documents, the standard terms have been preferred.

Since the English language can prove ambiguous on a par with ASN.1 semantics, a few comments on terminology may not be superfluous: we shall denote by *round brackets* the characters "`(`" and "`)`" also called parentheses. We call *square brackets* the characters "`[`" and "`]`", sometimes denoted by the shorthand (and deceptive!) 'bracket' in the literature. And finally we shall write *curly brackets* for the characters "{" and "}" also denoted by 'braces' elsewhere. This choice was motivated not only by the fact that these are the terms we usually use (!) but also and mainly because we found that their visual-sounding nature would be much helpful for the non-native English speaking readers among whom we can be counted.

After much hesitation about the structure to be given to the semantic rules, we went for the modal 'must' instead of the more formal ITU-T-like 'shall'[7]. We hope that the readers who are already familiar with this style will not find it too confusing.

Since English has been more 'an acquired taste than an innate gift' for me, people who influenced this text in one way or another would be

[7]Note, however, that strictly speaking, an ITU-T standard is no more than a 'Recommendation' after all, namely "a *suggestion* that something is good or suitable for a particular purpose". The layman may legitimately wonder whether finding 'shall' instead of 'should' in a recommendation could not constitute a contradiction in terms...

too numerous to mention here; they know who they are (some of them may at least!). Thanks to: Margaret, John and Savash for a few tips, the Association Georges Perec for the translated material (I'm not quite there yet...) and the reviewers on both sides of the Atlantic. Last but definitely not least, many thanks to Olivier for his continuous support and insightful explanations throughout the translation process.

Philippe Fouquart
Paris, France
October 1999

Part I

Introduction and History of the Notation

Chapter 1

Prologue

"Mr. Watson, come here; I need you!"

Alexander G. Bell, 10 March 1876.

Melville Bell, teacher of elocution at the University of Edinburgh acquired a worldwide reputation not only as an expert on correct pronunciation but also as the inventor of 'Visible Speech' [Bel67], a written code[1] providing a universal alphabet — *some may say an abstract notation* —, describing the precise positions and the respective actions of the tongue, the throat and the lips while speaking (see Figure 1.1 on the following page). This code published in 1867 [Bel67] was initially meant to make foreign languages — *the abstract syntaxes* — pronunciation easier since it provided a way to link one language to another. It finally proved more useful for teaching diction to deaf people.

In 1862, 15-year-old Alexander Graham, Melville Bell's son[2], uttered strange sounds that his father had written in 'Visible Speech' while he was out of the room — *using the speech basic encoding rules, Alexander associated a transfer syntax with the abstract syntax his father specified.*

A year later, Alexander and his older brother Melly built a talking machine out of a fake skull of gutta-percha filled with cans, rubber, and a lamb's larynx to make up the vocal parts. They then blew through it, making it cry out "*Ma-Ma!*".

[1]The set of characters can be found at http://www.indigo.ie/egt/standards/csur/visible-speech.html.

[2]A short story of Graham Bell and some photos are available at:
http://www.garfield.k12.ut.us/PHS/History/US/1877/inv/alex/default.html.

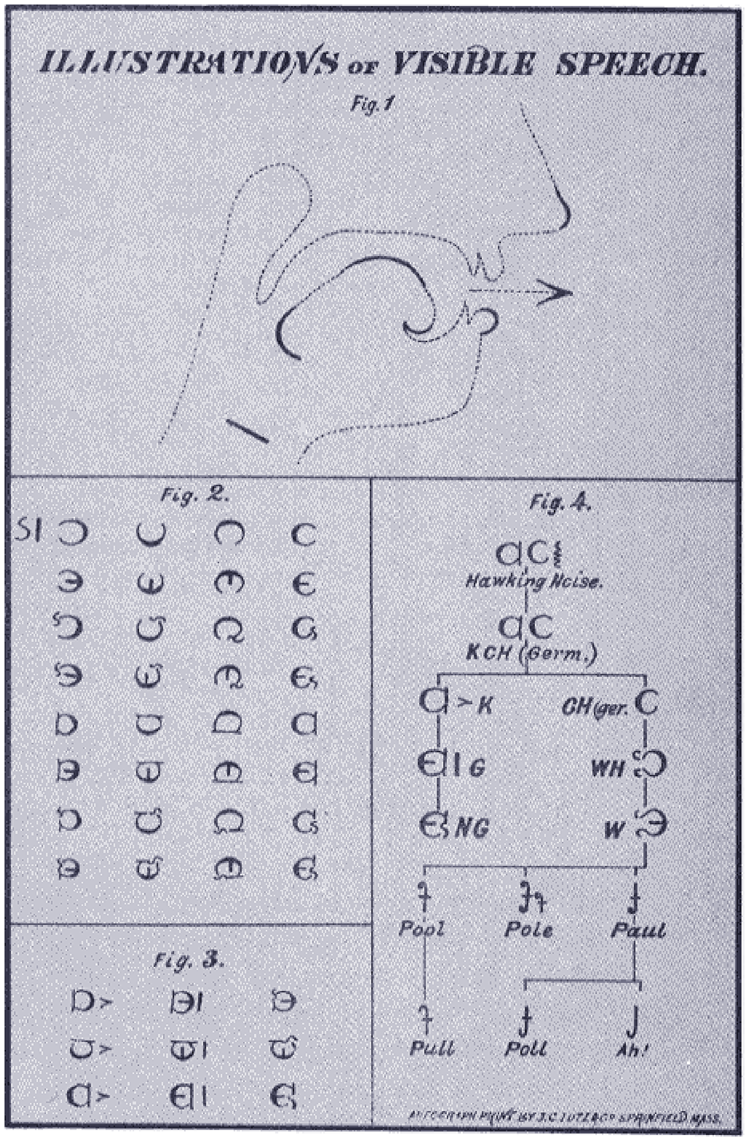


Figure 1.1: The three organs represented by 'Visible Speech': the lips (*Fig. 2*), the tongue (*Fig. 3*) and the throat (*Fig. 4*)

During the following years, Alexander helped his father teach the 'Visible Speech' code, particularly at institutes for deaf people, but he also took to studying vocal physiology. This work incited him to investigate ways to reproduce vocal sounds so that these could be transmitted electronically.

Alexander built up devices in which a first series of blades — *the encoder* — vibrated in resonance with basic sounds emitted by speech; these emitting blades drew to resonance a second series of blades using an electromagnet — *the decoder.* In fact, this work was more related to experiments of transmitting sounds both ways on a single line using a 'harmonical' system.

In 1874, Alexander invented the 'phonautograph'[3] — *another kind of receivor* — a device that could transcribe sounds into concrete signs: when words were pronounced, the ear membrane of a dead person vibrated and moved a lever on a tinted glass.

In July 1874, as Alexander wondered whether such a membrane could produce an electrical current whith an intensity that would vary according to the sound waves, he drew out the basics of the telephone in principle. It was patented in March 1876 at the U.S. Patent Office under the reference 174,465; it was modestly entitled "Improvements in Telegraphy" [Bro76].

On the 10th of March 1876, after several unconclusive attempts, Alexander uttered in his device the historical words, "Mr. Watson, come here; I need you!", which were heard by his workmate Thomas A. Watson in the receiver located in a distant room and set up the first phone conversation ever in history.

That is how the development of an abstract syntax notation brought about this amazing invention that changed the world...

[3] http://jefferson.village.virginia.edu/~meg3c/id/albell/ear.2.html

Before we leave this historical prologue and plunge into the core of the matter, we suggest the reader to stop off at this stage for a while, and ponder over this allegory of communications between heterogeneous systems...

Chapter 2

Utilitarian introduction to ASN.1

Contents

'Just to give you a general idea', he would explain to them. For of course some sort of general idea they must have, if they were to do their work intelligently – though as little of one, if they were to be good and happy members of society, as possible. For particulars, as every one knows, make for virtue and happiness; generalities are intellectually necessary evils. [...]

'I shall begin at the beginning', said the D.H.C., and the more zealous students recorded his intention in their note-books: *Begin at the beginning.*

Aldous Huxley, *Brave New World.*

Adopting an approach somewhat inspired by maieutics, we propose to draw out the reasons that lead to the definition of a notation such as

ASN.1. This chapter requires no prior knowledge of the 7-layer OSI model (the way ASN.1 finds its place in this model is described in the next chapter); a few notions about programming languages' grammars may prove useful for the first part.

This chapter introduces the underlying concepts and fundamental principles of ASN.1 used, sometimes implicitly, throughout this book. The main concepts of the notation are thoroughly described in Chapter 5.

2.1 A diversity of machine architectures

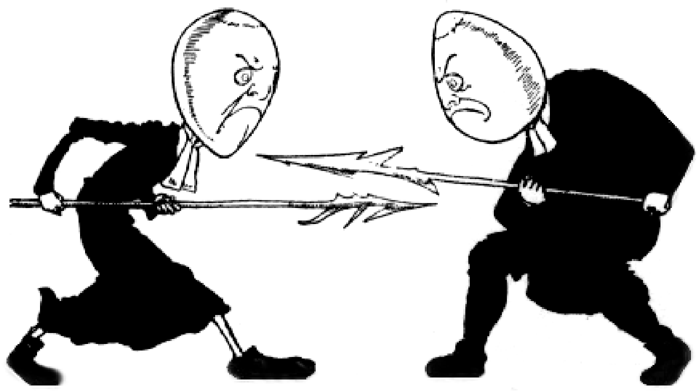
History has proved that data information is crucial, especially for cultural exchanges at first and then for developing an international economy. And more and more computing applications of all kinds are exchanging ever more complex data. Only think about banking exchange applications or airlines that have to manage bookings, staffs, routes, planes, maintenance, etc. or even mail order companies, which need to transfer data from their headquarters to their store houses, to make out the tremendous complexity of the data structures involved.

Contrary to common belief, these exchanges, though initially sped up by information technologies and then reinforced by the increase of telecommunication networks, have not simplified their structures. Indeed, networks and computers constitute a more and more heterogeneous world: computers can have different internal representation modes for the data they store. For example, a great majority uses the ASCII encoding[1], but some mainframes, such as IBM's, handle EBCDIC encoding. Most of PCs use 16 or 32-bit memory-word in two's complement arithmetic but some mainframes, sailing out of the mainstream, use 60-bit word in one's complement arithmetic for instance.

A much more insiduous difference is the following: x86 Intel chips, for example, order the bits of a byte from right to left whereas Motorola does just the opposite. It makes Danny Cohen [Coh81][2] call Intel a 'little

[1]Neither ASCII nor EBCDIC alphabets nor one's nor two's complement will be described here. They are used only for the sake of illustration. These notions are not necessary for understanding this chapter and can be found in [Tan96] for example.

[2]It is worthwhile noticing that this article brings us back to 1981, that is, before the creation of ASN.1 (see Section 6.3 on page 60). Incidentally, it ends with the visionary sentence: "*Agreement upon an order [of bits] is more important than the order agreed upon*". Isn't ASN.1 this final agreement, after all?



The Annotated Gulliver's Travels
Illustration by Willy Pogßny (1919) ; Isaac Asimov Editor
Clarkson N. Potter, Inc., New York (1980)
(courtesy of Random House, Inc.)

Figure 2.1: Battle between big Endians and little Endians!

Endian' (the number 0 low-weighted byte is on the right-hand side) and Motorola a 'big Endian'[3], referring to theological arguments between Catholics and Protestants in Swift's *Gulliver's Travels* as to whether boiled eggs should be broken at the top or the bottom! (See Figure 2.1.) This question does not come up only for eggs: [Tan96] mentions the case of an unfortunate 'stegosaurus' that once transferred from a 'little Endian' to a 'big Endian' machine, ended up with an exotic name and a length of 167,772 meters!

To prevent the 'big Endians' manufacturers (or the 'little Endians' manufacturers if you are on the 'big Endians' side') from being banished to remote Blefuscu Island[4] where communicating with the 'little Endians' is no longer possible, the latter would have to adopt the former's convention. But we know that, in order to avoid any compatibility malfunctions in their own products or even to preserve their position of hegemony in the market, few manufacturers would be inclined to sign such an armistice. And it is highly unlikely — some of the latter will definitely not regret it! — that one day some standard would define once for all an international internal data representation mode...

[3]Another example: the IP protocol is big Endian, the VAX computers are little Endians.

[4]In Swift's book, Blefuscu stands in for Louis XIV's 'so very Catholic France'!

```
typedef struct Record {
  char name[31];
  int  age;
  enum { unknown=0;
         male=1;
         female=2 } gender;
   } Record;
```

(a) *C* code

```
type record = {
  name   :  string;
  age    :  num;
  gender :  t_gender }
and t_gender   = Unknown
               | Male
               | Female
```

(b) *Objective Caml* code

Figure 2.2: Data structure in two computing languages

2.2 A diversity of programming languages

This heterogeneousness of machine architectures goes along with a great diversity in programming languages. For example, some data may be represented by an array of integers in one language and by a list of integers in another. Besides, a language may use its own representation in the internal memory: in *C* language, for instance, the '`\0`' character is added at the end of a string and programmers are free for their own defined boolean values because none exists by default.

For example, a data structure including a name declared as a 30-character string, a positive integer for the age, and the gender (male, female or unknown) may be represented as in Figure 2.2, in two very different ways in *C* and *Objective Caml*[5].

Note in particular that:

- the field `name` ends with a `NULL` byte in *C* but not in *Objective Caml*;
- the `name` field size is not limited in *Objective Caml*;
- nothing is known about the internal representation of `age` (one's or two's complement?...);
- the default values (provided they exist) of the enumerated type variables are unknown in *Objective Caml*.

[5] *Objective Caml* is a language of the *ML* family; it can be downloaded from http://caml.inria.fr.

These remarks bring up the following questions:

- do we have to keep the terminating null byte when transferring data? Does the *C* program have to skip out these values? Or is it up to the *Objective Caml* program to add it before transferring?
- How can we transfer the gender of a person without one of the three identifiers as a string (which can be arbitraly long) while keeping the 'semantics' (that is, the common acceptance of the identifier)?

The reader should now be convinced of the necessity of a transfer notation together with the need of conversion programs to and from this notation.

2.3 Conversion programs

Bearing in mind the variety of internal data representations we have just described, how does one make n machines 'converse'?

We could decide, as in Figure 2.3(a) on the following page, that every machine knows the internal data representation of its $(n-1)$ neighbors so that the data formats is agreed to be the receiver's or sender's[6]. Should the dialog take place both ways, this would require $n(n-1)$ conversion programs. This kind of communication has the undeniable advantage of translating data only once during the transfer. Although it could have been a decisive argument when processing time was a limiting factor, it is no longer relevant for today's data transfer.

We could also make up a common transfer format that would need to be agreed upon among different machines (or even better, standardized at an international level...). From Figure 2.3(b) on the following page it follows that n encoders and n decoders ($2n$ conversion programs altogether) would be needed, which is obviously smaller than $n(n-1)$ when n, the number of machines, is greater than 3.

In addition, if these $2n$ programs could be automatically generated, a significant gain would be obtained in terms of efficiency (no specific development) and reliability (no possible misinterpretation during the encoder/decoder implementation).

[6]The NIDL notation from Apollo Computer, which is described in Section 24.2 on page 490, is a good example of symmetrical communication.

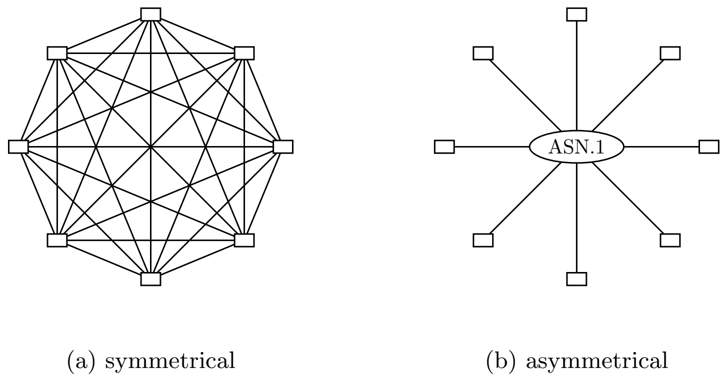


Figure 2.3: Two types of communications

2.4 The triad: concrete syntax, abstract syntax, transfer syntax

Now that the problematics has been set out, let us draw up a synthesis and introduce some definitions.

We call *concrete syntax*, the representation, in a given programming language, of the data structures to be transferred. It is *a* 'syntax' because it respects the lexical and grammatical rules of a language (*C* for instance); it is called concrete because it is actually handled by applications (implemented in this very language) and it complies with the machine architectures' restrictions. Two examples of concrete syntaxes have been given in Figure 2.2 on page 10.

In order to break free of the diversity of concrete syntaxes mentioned above, the data structures to be transmitted should be described regardless of the programming languages used. This description should also respect the lexical and grammatical rules of a *certain* language (guess which language) but should remain independent from programming languages and *never* be directly implemented on a machine. For these reasons, we call *abstract syntax* such a description and *Abstract Syntax Notation* or *ASN*[7] the language whereby this abstract syntax is denoted.

[7]The meaning of 1 in ASN.1 is given in Chapter 6.

Though independent from programming languages, the abstract syntax notation should be at least as powerful as any language's datatype[8] formalism, that is, a recursive notation that allows building complex data types from basic types (equivalent to the `string`, `int`, `enum`... *C* types for instance) and type constructors (equivalent to `struct`, `union`... in *C*).

Many different messages can be exchanged between applications. The abstract syntax would then describe in a more condensed way the whole set of these messages. For this, the abstract syntax is defined by means of a grammar that data to be transferred should respect. Otherwise said, one should bear in mind the two levels of grammars [ASU86]: first, the grammar of the 'ASN' abstract syntax notation itself; second, the abstract syntax, which is also a grammar[9] and is defined using the former.

This abstract syntax notation must be formal to prevent all ambiguities when being interpreted and handled by computing tools as described at the end of this chapter. Drawing a parallel with language theory, the abstract syntax notation can be seen as a BNF (or Backus-Naur Form) that allows one to write specific *grammars*, which are called abstract syntaxes in a data transfer context. In a nutshell, ASN.1 is *not* an abstract syntax[10] but a language to describe them.

Referring now to a more general concept of 'circulation of immaterial streams', it is only the material component of a piece of information that is involved in the communication. It is not the *sense* that is conveyed but its material representation which, in essence, is of a physical nature. In other words, the form is conveyed but not the content. It is only when this representation arrives on the receiver's side that it takes on its full meaning.

For the same reason, the abstract syntax defines precisely the data structure but says nothing about the associated semantics, viz., the interpretation of these data by the application (thus by its programmer): what meaning should we associate with a TRUE boolean value? What

[8]We should stress on the word *datatype* because an ASN.1 specification is only a static description of the data; it cannot describe the operations to be applied to these data.

[9]The language defined by this *second* grammar is the set of all the values that can be transferred.

[10]This a very common (though harmless) confusion made by ASN.1 users.

am I supposed to do if no value is assigned to a field? Many questions to remain unanswered because they do not fall within the competence of data transfer but are resolved when the data is used by the application[11]. The semantic part is described, if necessary, by means of comments within the abstract syntax or using an explanatory text associated with it.

We have seen in Section 2.3 on page 11 and in Figure 2.3(b) on page 12, the advantages of a data transfer which is independent from the machine architectures. As for the data received as byte streams or bits, they comply with a syntax called *transfer syntax* so that these streams could be properly recognized by the peer machine.

Of course, this transfer syntax thoroughly depends on the abstract syntax, since it sets up how the data should be transmitted according to this abstract syntax. In fact, the transfer syntax structures and orders the bytes (the 'formant') that are sent to the other machine (this process is sometimes called 'marshalling'). But contrary to the abstract syntax, it is a physical quantity and, because of that, it should take into account the ordering of the bytes, the weight of the bits, etc.

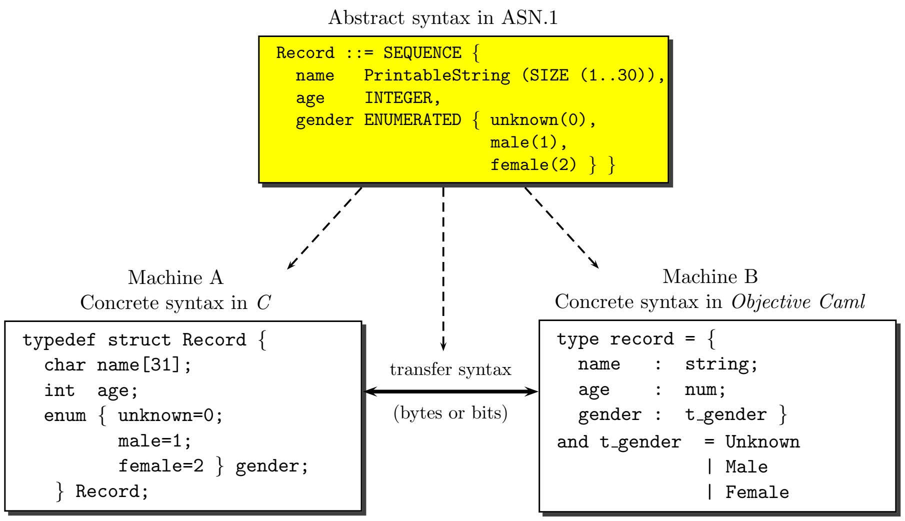
Different transfer syntaxes can be associated with a single abstract syntax. This is particularly interesting when the throughput increases and makes more complex encoding necessary: in such a case, however, it is possible to change the transfer syntax without changing the abstract syntax.

If we now come back to the example in Figure 2.2 on page 10, we end up with the four syntaxes of Figure 2.4 on the next page (one abstract syntax, two concrete syntaxes and one transfer syntax). The dashed arrows set out the links that exist between the abstract syntax and the concrete syntax.

From a single ASN.1 data description, we can derive automatically as many concrete syntaxes (i.e. in as many programming languages) as necessary, and as many procedures implementing the transfer syntax in encoders (which encode the data into a bit or byte stream) and decoders as we want.

As described in Chapter 22 (more particularly in Figure 22.2 on page 465) it is an ASN.1 compiler that carries out the automatic generation for us according to the dotted arrows in Figure 2.4 on the next page, thus sparing considerable effort and meanwhile making it possible

[11]Nevertheless, we will see in Chapter 15 that since 1994, information object classes can formalize strong semantic links which can be really convenient for the users.



```
Record ::= SEQUENCE {
  name   PrintableString (SIZE (1..30)),
  age    INTEGER,
  gender ENUMERATED { unknown(0),
                      male(1),
                      female(2) } }
```

```
typedef struct Record {
  char name[31];
  int  age;
  enum { unknown=0;
         male=1;
         female=2 } gender;
   } Record;
```

```
type record = {
  name    :  string;
  age     :  num;
  gender  :  t_gender }
and t_gender   = Unknown
               | Male
               | Female
```

Figure 2.4: An example of syntax triad

to inter-connect any number of machines (as in Figure 2.3(b) on page 12). The compiler should be implemented with some *encoding rules*, which describe the links between the abstract syntax and the transfer syntax[12].

[12]Many standards (including [ISO8822] on the Presentation service definition for example) sometimes do not distinguish the notion of a transfer syntax from that of encoding rules. We may equally use one or the other in the next chapters whenever it remains unambiguous.

Chapter 3

ASN.1 and the OSI Reference Model

Contents

Should the intermediate ranks of a monarchy be removed, there would be too far from the monarch to the subjects; soon afterwards one would see only a despot and some slaves: preserving a graduated scale from the ploughman to the potentate equally concerns all men, whatever their ranks, and can be the strongest support of a monarchist constitution.

Beaumarchais, *Le Mariage de Figaro*.

Having described the benefits of ASN.1 a priori, we now come to its role in the OSI model. After a short introduction on the 7-layer OSI model, we shall discuss in greater length the layers 6 (called *Presentation*) and 7 (*Application*) in the context of which the notions of transfer syntax and abstract syntax introduced in the previous chapter will be re-defined.

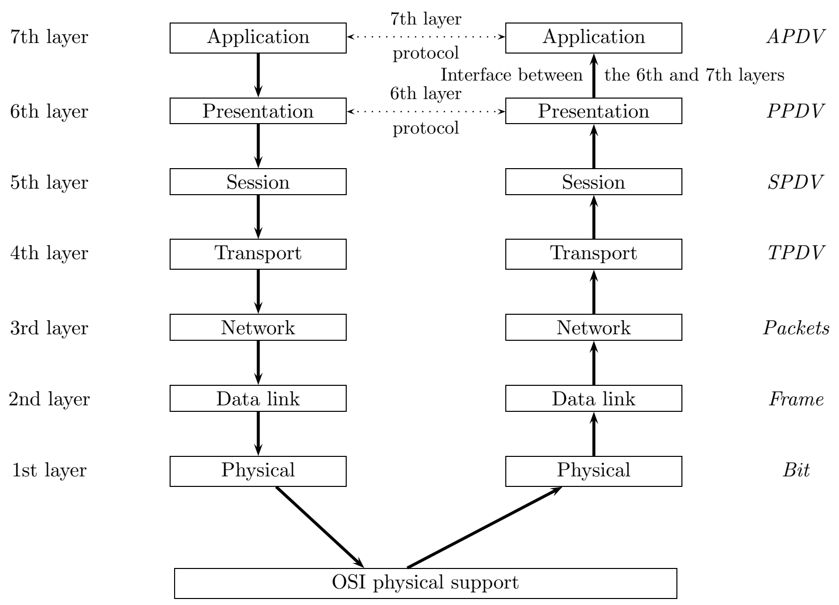


Figure 3.1: 7-layer OSI model. Example of information streams

The main point of this chapter is to put ASN.1 in its historical background. ASN.1, however, should not be taken as being limited to this very domain: though historically related to the OSI model, we shall see in Chapter 7 that ASN.1 is actually used in many applications that are not based on this 7-layer model.

3.1 The 7-layer OSI model

Telecommunication networks are modeled by a graded order of hierarchical layers to make standardization and conception easier. The role of each layer is to provide services to the layer above, relying on those offered by the layer below. The higher the layer, the more services it can offer (but the more abstract their definitions are!) and the less important the way data are conveyed becomes. The most famous network architecture is the *Open Systems Interconnection model* or *OSI model*, which features 7 layers as shown in Figure 3.1. It is standardized at ITU-T under the name “Recommendation X.200” and at ISO under the reference [ISO7498-1].

The OSI model goes from the lowest level of signalling techniques up to high level interactions between specific applications[1]. In the increasing order, i.e. from bottom to top in Figure 3.1 on the preceding page, these seven layers are:

- the *Physical layer*, in charge of the interface between systems and the physical support, deals with the transmission of bits. It is concerned with the voltage required to represent a single bit, the duration of a bit representation in milli-second, the transmission both ways if needed and other issues of electrical and mechanical matter;

- the *Data Link layer* ensures the transmission of information using error correction and splitting the data into frames (set of bits); since the physical layer transmits bit streams without knowing its structure, it is down to the Physical Layer to mark up the beginning and the end of the frame;

- the *Network layer* manages the routes (statically and dynamically) to be taken by the packets (set of frames) to go from one equipment to another;

- the *Transport layer* ensures that the data are transmitted at the right throughput and that they are correctly collected on the receiving side. It also makes all technology or equipment changes transparent to the Session layer;

- the *Session Layer* manages the exchange sessions between two machines: dialog tokens indicating which is emitting, synchronization to carry on a transmission where it was interrupted...;

- the *Presentation layer* is interested in the syntax of the data to be transmitted. It is in charge of encoding them according to given rules that have been agreed on beforehand;

- the *Application layer* is concerned with frequently used application protocols such as file transfer, electronic mail, remote login...

The layer n of a machine can speak to the layer n of another providing the dialog complies with certain rules and conventions specific to this

[1]But little interest has been shown so far to include the layers 8 (*Financial*) and 9 (*Political*) in the standardized model!

level. The collection of these rules constitutes the 'n-layer protocol[2]'. The data transmitted by an application that is running on the first machine should go through the layers down to a medium link. A header (and an ending mark if required) is added to the data each time it crosses a new layer. In other words, the $n + 1$ layer states out how the gap in the n-level data should be filled.

For each layer of the model, two international standards can be found: the first describes the service provided by this layer (and defines the terms and the notation it uses), the second describes its protocol, i.e. the rules any implementation should respect.

We shall not proceed further with our overview of the OSI model. The interested reader should find enough material to satisfy any avid curiosity in [Tan96], [PC93], [Bla91] or [Lar96]. We will go now into more detail about the top two layers (called Presentation and Application layers) of the OSI model.

3.2 The Presentation layer

The Presentation layer is the sixth layer of the OSI model; it is described in the standards [ISO8822] and [ISO8823-1] (among others). It assumes that a 'route' exists between two machines, that it provides the adequate quality of service and that all the functions potentially needed are available in the low level layers.

Its main role is to ensure the encoding and decoding of the data. As previously seen in Section 2.1 on page 8, the data representation (integers, real numbers, strings...) depends on the machine so that a common representation is required for exchanging the data. The presentation layer provides the data to be transmitted with this structure but is not concerned in the semantics of the information.

To make two speakers converse, we need first to share a common knowledge (a dictionary) and second, we have to agree on a language before starting a conversation using preliminaries such as:

> — *Do you speak English? Parlez-vous français ? Te parau anei 'oe i te reo Tahiti ?*
> — *Je parle français. I speak English.*
> — *Let's speak English then!*

[2]Like those of etiquette and precedence to be respected in ceremonies or official protocols!

Likewise, two systems that communicate should negotiate a common encoding (BER, PER...) before transmitting the data. It is the role of the Presentation layer to negotiate this encoding and to actually encode the information exchanged between the two applications that cooperate.

For doing so, the Presentation layer makes use of the following concepts:

- the *abstract syntax*, which defines the generic structure of the data (the various types of data involved) and constitutes the framework on which relies the dialog with the Application layer (for example, the data include a component of type boolean and another one of type integer, which is optional);

- the *concrete syntax* is local and defines the data representation in the local system (for example, a *C*-language implementation of the types above mentioned);

- the *transfer syntax* defines the data representation exchanged between the respective Presentation layers of the two systems thanks to their Session layers;

- the *encoding rules* provide a means of going from the local concrete syntax to the transfer syntax and reverse (for example, all the data are encoded as triplets ⟨data type, data length, data value⟩; boolean values are encoded on a byte with the value 1 or 0...).

Using these notions, we could say that the Presentation layer provides the following services to the Application layer:

- negotiation[3] of the transfer syntax (a means of choosing one at the beginning of the dialog and a way to change it);
- identification of a collection of transfer syntaxes (i.e. several ways of representing the abstract syntax);
- translation, using the encoding and decoding rules of the concrete syntax (internal representation mode), into the transfer syntax (external representation mode) and reverse;
- association of a negotiated transfer syntax to the abstract syntax adopted within the applications;
- access to the Session layer's services.

[3]If the transfer is in 'no connection' mode, the transfer syntax is not negotiated; the sender chooses it arbitrarily.

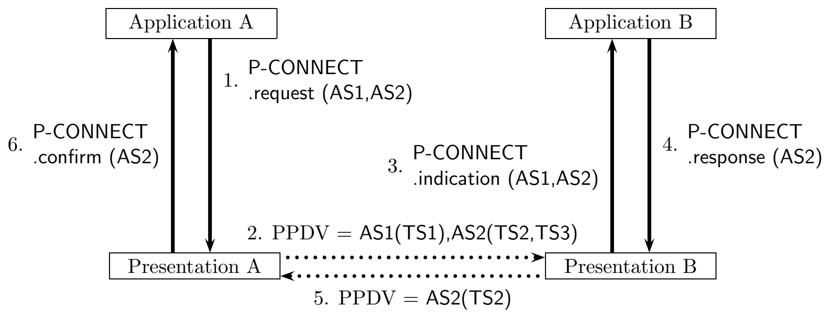

Application A
Application B
1. P-CONNECT .request (AS1,AS2)
6. P-CONNECT .confirm (AS2)
3. P-CONNECT .indication (AS1,AS2)
4. P-CONNECT .response (AS2)
2. PPDV = AS1(TS1),AS2(TS2,TS3)
Presentation A
Presentation B
5. PPDV = AS2(TS2)

Figure 3.2: Presentation context negotiation

The Presentation layer guarantees the informational content of the Application layer's data. The cooperating applications are in charge of determining all the abstract syntaxes they use for communication and of informing the Presentation layer. As it knows all the available abstract syntaxes, the Presentation layer is in charge of the choice of the transfer syntaxes that are mutually acceptable.

Figure 3.2 presents in more detail the way an application should invoke the services of the Presentation layer to negotiate the abstract syntax and the transfer syntax to be used during the data transfer.

1. Application A sends a P-CONNECT.request primitive[4] to its Presentation layer stating the names (AS1,AS2) of the abstract syntaxes according to which it may operate the transfer. Each name of an abstract syntax is actually a series of numbers called an *object identifier*[5], which can identify the abstract syntaxes universally: indeed, we should bear in mind that it is an open architecture, it should accept all sorts of machines, all sorts of abstract syntaxes and all sorts of transfer syntaxes.

[4]A *primitive* is a function which a user or an entity can use to access a service provided by a layer of the OSI model. A primitive asks a service to carry out an action or to give account of a measure taken by the entity with which the communication is set.

[5]It is a value of the ASN.1 type `OBJECT IDENTIFIER` described in Section 10.8 on page 153.

2. The Presentation layer associates several transfer syntaxes with each abstract syntax and encodes for the Session layer a Presentation Protocol Data Value (PPDV)[6], which is sent to the Presentation layer of the other system. This PPDV contains the names of the available abstract syntaxes.

3. The Presentation layer B receives this PPDV and sends back a P-CONNECT.indication primitive to its own Application layer indicating the abstract syntaxes available in Application A.

4. Application B answers with a P-CONNECT.response primitive indicating the names of the abstract syntaxes that can be used for the transfer[7] (only AS2 here).

5. The Presentation layer B receives the primitive and sends a PPDV stating the transfer syntax which is used with each transfer syntax that have been agreed on.

6. Finally, the Presentation layer A receives the PPDV, examines the proposed transfer syntax and if it accepts this, sends a P-CONNECT.confirm primitive to Application B.

Note that the Presentation layer is not involved in the determination of all the abstract syntaxes available for the applications. In general, there may be more than one combination abstract syntax/transfer syntax. One abstract syntax can be represented by one or more transfer syntax(es)[8]; it is also possible to use one transfer syntax to represent more than one abstract syntax[9]. The result of the negotiation for the abstract syntax/transfer syntax combination is called a *presentation context*[10]. Other contexts can be negotiated dynamically during communication.

[6]The Presentation Protocal Data Unit (PPDU) that matches this PPDV (i.e. its type) is defined on page 361.

[7]If no abstract syntax is known by Application B, the connection stage is denied.

[8]It occurs for example when the application operates a data encryption: the two application contexts are negotiated at the connection and used according to the application's needs.

[9]It occurs for example when documentary database consultations require searching phases with interactive dialogs followed by the transfer of the selected documents (see [T.433] for more detail about the transfer service and the structured document remote handling standards): two abstract syntaxes are encoded with the same transfer syntax.

[10]The presentation contexts are identified by integers that are even numbers for

At the end of the initial negotiation the system has a whole set of presentation contexts at its disposal among which it can, at any time, choose the appropriate context for the exchange to operate.

The data that come from the Application layer are encoded with respect to the presentation context involved. When only one context is specified, they are directly encoded (simple encoding), otherwise all the application data including the embedded data are preceded by the appropriate context identifiers (complete encoding).

This presentation is not meant to be exhaustive. This and many related issues are discussed in comprehensive texts on the subject quoted at the end of Section 3.1 on page 18. In practice, this negotiation mechanism is hardly ever implemented because the abstract syntax and the transfer syntax are rarely negotiated for communication. However, the description above is important to understand (partially at least) the use of the `OBJECT IDENTIFIER` type (see Section 10.8 on page 153) or the presentation context switching types such as `EXTERNAL`, `EMBEDDED PDV` and `CHARACTER STRING` described in Chapter 14.

3.3 The Application layer

The seventh and last layer of the OSI model is called Application layer. As the highest layer, it is for the communicating application the only access to the OSI environment and, as such, it provides all the services that can be used directly by the application.

When referring to the OSI model, we call every communication aspects of an application, an *application entity*[11]; application entities use application protocols and presentation services to share information.

The data structure of each application are sent as Application Protocol Data Values (APDV, see Figure 3.1 on page 18) specified in ASN.1. Each time an application wants to transfer data, it provides the corresponding APDV to the Presentation layer together with its ASN.1 name. Referring to the ASN.1 definition the Presentation layer knows the type and length of the data components and the way these should be encoded to be transmitted. On the other side of the connection, the Presentation layer analyzes the ASN.1 identifier of the expected data

one entity and odd numbers for the other to prevent overwriting when allocating new contexts.

[11]For simplicity's sake, we shall nonetheless use the term 'application' in the rest of this text.

structure and then knows how many bits belong to the first component, to the second etc. With this information, the Presentation layer can operate all the necessary conversions to provide the data according to the internal format of the receiving machine.

ASN.1 is the only representation used for OSI applications since ISO demands that all the data exchange between the Application and the Presentation layer should be described with an ASN.1 abstract syntax. As far as the OSI model is concerned, ASN.1 is mainly used for high-level layers (particularly because low-level layers existed before ASN.1), but this should not be considered as a restriction. If ASN.1 were more generally used on low level layers, sufficiently compact encoding would have to be used to prevent overloading the data.

Concerning the Application layer, such a structured and powerful notation as ASN.1 is necessary because it is not possible to gather bits in bytes[12] any more as it could be done for lower-level layers. Moreover, we cannot ask application designers to be perfectly aware of problems encountered only when encoding the messages in bits! The semantics of the transferred data depends on the application and, because of that, it only concerns the Application layer. This semantics is partly described with ASN.1 in an abstract syntax definition, and the rest is described in comments or documented with the ASN.1 specification.

Some applications were so widespread that standards were agreed upon to prevent every company from developing specific programs and thereby ensured the most general use for the defined protocols:

- ACSE (*Association Control Service Element* [ISO8650-1]) is designed to set a connection called 'association' on the Application layer between two application entities and ensures that this terminates with no loss of information.

- ROSE (*Remote Operation Service Element* [ISO9072-2] and [ISO13712-1]) makes the existence of a communication between remote processes transparent for the application programmer and enables operation remote calling;

- RTSE (*Reliable Transfer Service Element* [ISO9066-2]) provides a transparent APDU transfer service, which is safe and generic (restarting the transfer where it was interupted if needed);

[12]In practice, a significant number of protocols (in the ISDN domain, for example) is still specified using cross-array for grouping bits. Experience shows that the cost in terms of tests and validation of such informally described protocols is significantly high (see Section 23.2 on page 480).

- CCR (*Commitment, Concurrency, and Recovery* [ISO9805-1]) coordinates safely the interactions between multiple sites even in breakdown situations and is used when high-level reliability is required;
- MHS (*Message Handling System* [X.400]) is an e-mail system.

The list of application services is not exhaustive; these can be combined to build up more complex units.

3.4 The OSI model in the future

The OSI model, although still relevant for today's computing architecture, is sometimes considered to be phasing down because:

- it is mainly used by those who cannot avoid it (the national standardization organizations that are on the ISO committees or the ITU-T's operators for example);
- the standard documents are expensive and difficult to obtain from the national standardization organizations[13];
- the number of layers is excessive, for it makes the model too complex (still note that an efficient application of an OSI protocol can limit the size of the headers and ending blocks induced by the 7-layer model to around 10 % of the overall size);
- ASN.1 remains an artificial addition to the model: the model had originally no Application layer and was not meant to: the Presentation layer would have specified document format common for all the applications and the Session layer would have been used to make sure a copy of this document was properly maintained on each side;
- it provides standards for services that are insufficiently tested before their publication;
- it is not applicable for new types of networks such as ATM;

[13]However, note that the JTC 1 (see Section 6.1 on page 54) asked the ISO Council for distributing the various versions of the standards on the Web. The users cannot wait for the answer!

- it is much less established and active in TCP/IP protocols (even though these are equivalent to the layers 3 and 4 and though the OSI stack offers a set of functionnalities that are more advanced and sophisticated than TCP/IP) and it does not offer reliable tools both affordable and adapted to the client's specific needs.

We shall not discuss the model's future in greater length[14]; it should be noted that ASN.1 is one of the concepts that should live through, particularly because it is more and more used in protocols outside the OSI world (see Chapter 7). However, to ensure that the notation keeps up its position, the standard should get rid of its few ambiguous parts, sometimes awkwardly put, and the new concepts that will be introduced should be easier to manipulate and should prove useful for the specifier. This would prevent the notation from being considered as the private domain of a few, perhaps suspected to be disconnected from reality and ill-at-ease when it comes to follow the technical changes. It is definitely today's willingness of the ASN.1 working group. Long live ASN.1!

[14]Other drawbacks of using the OSI model are listed in [Tan96, Section 1.6].

Chapter 4

Your first steps with ASN.1

Contents

"Why, then," inquired the Sirian, "do you quote the man you call Aristotle in that language?"
"Because," replied the sage, "it is right and proper to quote what we do not comprehend in a language we least understand."

Voltaire, *Micromegas.*

In the previous two chapters, we have exposed the main principles of data transfer, particularly in the context of the OSI model. We now put this into practice in the study of a 'real life' example where we tackle the problem of specifying data transfer and how to describe these data in the abstract syntax notation.

The ASN.1 entities used in this chapter are fairly simple but they will make the concepts described in the next chapter easier to introduce. This tutorial mainly aims at helping the reader get familiar with ASN.1 concepts and syntax. For doing so, we start from the informal description of a module and draw out its ASN.1 equivalents.

4.1 Informal description of the problem

Suppose a company called MicromegAS owns several sales outlets linked to a central warehouse where the stocks are maintained and deliveries start from. It has been agreed that this network should be organized using OSI solutions and a client-server model[1]:

- the orders are collected locally at the sales outlets;
- they are transmitted to the warehouse, where the delivery procedure should be managed;
- an account of the delivery should be sent back to the sales outlets for following through the client's order.

The following information is available:

- an order should include the client's identifier, i.e. name and address (by default, he/she lives in France);
- the order's identifier should be unique;
- an order can feature several items identified by a code described with a label;
- depending on the item, the quantity should be indicated in units, meters or kilograms; in the last two cases, it must be represented by a positive number of millimeters or milligrams (decimals are not valid);
- the prices should be indicated in Euros (the European currency unit), with a cent precision (each price will be a whole number of cents);

[1]The client-server model consists in interrogating a remote application and in collecting the answers. It uses the ROSE service described on page 80.

- the method of payment is associated with each order: check, credit card, cash. For a check, the check number should be noted; for a credit card, its type (CB, Visa, Eurocard, Diners or American Express), and its number together with an expiry date should be provided;

- the delivery report should be sent back to the warehouse; it should include for each ordered item, the quantity that can actually be delivered when the order is made, depending on the stocks.

4.2 How should we tackle the problem?

Contrary to most programming languages, ASN.1, as a specification language, prescribes no order when defining the entities. It is actually possible (even preferable for the sake of readibility and comprehension) to reference the data types before their definition in the specification. We take advantage of this flexibility to apply a top-down approach to our problem (from general to particular): first, we consider the problem globally to describe the most general data types and put off the definition of the types that are referenced in the general data types; second, we concentrate on those more specific types to complete the whole specification.

This approach is very much similar to the famous four principles of Descartes' *Discourse on the Method* on which we may reflect for a moment before we carry on further with the case study:

> And as a multitude of laws often only hampers justice, so that a state is best governed when, with few laws, these are rigidly administered; in like manner, instead of the great number of precepts of which logic is composed, I believed that the four following would prove perfectly sufficient for me, provided I took the firm and unwavering resolution never in a single instance to fail in observing them.
>
> The first was never to accept anything for true which I did not clearly know to be such [...].
>
> The second, to divide each of the difficulties under examination into as many parts as possible, and as might be necessary for its adequate solution.
>
> The third, to conduct my thoughts in such order that, by commencing with objects the simplest and easiest to know, I might ascend by little and little, and, as it were, step by step, to the

knowledge of the more complex; assigning in thought a certain order even to those objects which in their own nature do not stand in a relation of antecedence and sequence.

And the last, in every case to make enumerations so complete, and reviews so general, that I might be assured that nothing was omitted.

We now focus on the orders taken at the sales outlets. An order can be divided into two parts: a header including the client's identification, followed by a block made of the various ordered items. In ASN.1, this can be written as follows (the ASN.1 keywords are spelt in capital letters):

```
Order ::= SEQUENCE {
  header  Order-header,
  items   SEQUENCE OF Order-line}
```

It can be read: 'an `Order` is a structure (`SEQUENCE`) with two components: the first called `header` (starting with a lower-case letter), denotes data of type `Order-header` (starting with an upper-case letter); the second called `items`, denotes a list (`SEQUENCE OF`) of data which are all of type `Order-line`'.

`Order` is an ASN.1 type; note it begins with a capital letter, it is followed by the "`::=`" symbol and that its definition does not end with a semicolon contrary to many programming languages.

4.3 Ordering an item: from general to particular

We now concentrate on the 'order header', which can be described as a structure with four components: order number, order date, client description and method of payment:

```
Order-header ::= SEQUENCE {
  number    Order-number,
  date      Date,
  client    Client,
  payment   Payment-method }
```

The order number has at least 12 digits. The order date should be described by an 8-character string, 2 for the day, 2 for the month, 4 for the year; this is indicated in the specification comments to make it easier to understand and prevent ambiguities on the meaning to be given to this value (when emitted or received) by the application:

```
Order-number ::= NumericString (SIZE (12))
Date ::= NumericString (SIZE (8)) -- DDMMYYYY
```

A client is described by name, postcode and town (and optionaly street and country):

```
Client ::= SEQUENCE {
  name        PrintableString (SIZE (1..40)),
  street      PrintableString (SIZE (1..50)) OPTIONAL,
  postcode    NumericString (SIZE (10)),
  town        PrintableString (SIZE (1..30)),
  country     PrintableString (SIZE (1..20))
                DEFAULT default-country }
default-country PrintableString ::= "France"
```

The '`OPTIONAL`' clause indicates that the component `street` is not necessarily transmitted. The clause '`DEFAULT default-country`' indicates that if the `country` component is not transmitted then the receiver should assign the default value `default-country` to the `country` field. This reference, which starts with a lower-case letter, denotes an ASN.1 value of type '`PrintableString`'. We could obviously have given the value `"France"` after the keyword `DEFAULT`.

Many times already, we have used in the specification a subtype constraint (`SIZE`) to limit the size of a character string. The subtype specifications are placed in round brackets after the type. They render the specification more precise and sometimes enable to improve the transmitting encoding. A constraint such as `SIZE (5)` imposes that the postcode should consist of exactly five characters and `SIZE (1..20)` calls for a 1 to 20-character string.

Let us now concentrate on the payment method. The client can choose between:

1. paying by check, in which case the sales outlet transmits the invoice number;

2. paying by credit card indicating the type of card, its number and its expiry date;

3. paying cash.

Note that we choose not to transmit the total amount of purchase assuming that it will be computed again since all the order-lines are received; here is another characteristic of ASN.1: it is not concerned with the operations applied to the data before or after reception. Such information, if needed, can be written in comments.

```
Payment-method ::= CHOICE {
  check          NumericString (SIZE (15)),
  credit-card    Credit-card,
  cash           NULL }
```

Each alternative is identified by a name (beginning with a lower-case letter) and denotes a value of a certain type. For the `cash` alternative, the only piece of information to be transmitted is whether this was chosen or not. Therefore we associate this alternative with the `NULL` type, which has only one possible value, `NULL`, whose encoding is inexpensive.

Using what we already know about ASN.1, we can write the type:

```
Credit-card ::= SEQUENCE {
  type        Card-type,
  number      NumericString (SIZE (20)),
  expiry-date NumericString (SIZE (6)) -- MMYYYY -- }
```

From the five types of credit cards of our informal description, it follows:

```
Card-type ::= ENUMERATED { cb(0), visa(1), eurocard(2),
                diners(3), american-express(4) }
```

Note that each identifier of the enumeration starts with a lower-case letter. The number in round brackets are important since it is *they* that are transmitted to the receiving application: the fields' labels are there only to help the reader understand the specification and to indicate the meaning associated with each enumerated value. We shall see on page 136 that these identifiers can be numbered automatically.

We come now to the order line, which is given by:

```
Order-line ::= SEQUENCE {
  item-code    Item-code,
  label        Label,
  quantity     Quantity,
  price        Cents }
Item-code ::= NumericString (SIZE (7))
Label ::= PrintableString (SIZE (1..30))
Cents ::= INTEGER
```

We prefer to call '`Cents`' the type of the `price` component rather than indicating in comments that the price of the article is an whole quantity in cents. Generally speaking, the type names and value identifiers must be chosen with great care: if the information is not in comments it will be kept by the ASN.1 tools and appear in the programs they generate (otherwise it is discarded).

Depending on the items, the quantity can be expressed in three different ways:

```
Quantity ::= CHOICE { units       INTEGER,
                      millimeters INTEGER,
                      milligrams  INTEGER }
```

In the next section, we show how principles that are normally down to the encoding rules can be taken into account on the abstract syntax level. It can be left aside if the reader would rather finish up with the last part of MicromegAS specification first (and go directly to Section 4.5).

4.4 Encoding and condition on distinct tags

The way the `Quantity` type has just been defined is in fact invalid: we cannot guarantee that a quantity expressed in millimeters will be interpreted in the same unit by the receiving application. Indeed, when encoded[2], each type is associated with a tag (suppose it is a number for the moment) to be transmitted just before the value so that the receiver could interpret properly the arriving data. As the three alternatives of the `CHOICE` have the type `INTEGER`, they will all have the tag `2`, which is by default the tag given by the ASN.1 standard to the type `INTEGER`. Therefore, we have to associate by hand a distinct tag with every alternative using square brackets before each type:

```
Quantity ::= CHOICE { units       [0] INTEGER,
                      millimeters [1] INTEGER,
                      milligrams  [2] INTEGER }
```

The same problem may have occured for the type `SEQUENCE` for which some components could have been optional as we shall see in Section 12.2 on page 218. The tags are also used to detect whether optional components are absent or present. In fact, all the `SEQUENCE` types introduced since the beginning of the chapter are not concerned with the problem.

[2]In this case study, a BER encoding is assumed.

However, if the `SEQUENCE` types are replaced by `SET` types, all the components should be tagged by hand (i.e. that of the specifier...) as in the following example:

```
Client ::= SET {
  name     [0] PrintableString (SIZE (1..20)),
  street   [1] PrintableString (SIZE (1..50)) OPTIONAL,
  postcode [2] NumericString (SIZE (5)),
  town     [3] PrintableString (SIZE (1..30)),
  country  [4] PrintableString (SIZE (1..20))
                 DEFAULT default-country }
```

Indeed, the `SET` type is also a structure but its components are not necessarily sent in the specification's order; through these tags the receiver has a means of determining each component received.

We shall see in Section 12.1.3 on page 213 that it is possible to ignore the problem of unicity of each tag putting the clause `AUTOMATIC TAGS` in the header of the ASN.1 module (as we will do for the `Module-order` module just below).

4.5 Final module

We conclude the specification of MicromegAS's data transfer with the delivery report sent by the sales outlets after receiving an order:

```
Delivery-report ::= SEQUENCE {
  order-number   Order-number,
  delivery       SEQUENCE OF Delivery-line }
Delivery-line ::= SEQUENCE { item     Item-code,
                             quantity Quantity }
```

All the types defined in this chapter are then grouped in the same *module*, which constitues the complete specification for the initial problem of the MicromegAS company presented in Section 4.1 on page 30:

```
Module-order DEFINITIONS AUTOMATIC TAGS ::=
BEGIN

Order ::= SEQUENCE {
  header  Order-header,
  items   SEQUENCE OF Order-line}
```

```
Order-header ::= SEQUENCE {
  number   Order-number,
  date     Date,
  client   Client,
  payment  Payment-method }

Order-number ::= NumericString (SIZE (12))

Date ::= NumericString (SIZE (8)) -- MMDDYYYY

Client ::= SEQUENCE {
  name     PrintableString (SIZE (1..20)),
  street   PrintableString (SIZE (1..50)) OPTIONAL,
  postcode NumericString (SIZE (5)),
  town     PrintableString (SIZE (1..30)),
  country  PrintableString (SIZE (1..20))
               DEFAULT default-country }
default-country PrintableString ::= "France"

Payment-method ::= CHOICE {
  check        NumericString (SIZE (15)),
  credit-card  Credit-card,
  cash         NULL }

Credit-card ::= SEQUENCE {
  type         Card-type,
  number       NumericString (SIZE (20)),
  expiry-date  NumericString (SIZE (6)) -- MMYYYY -- }

Card-type ::= ENUMERATED { cb(0), visa(1), eurocard(2),
                diners(3), american-express(4) }

Order-line ::= SEQUENCE {
  item-code    Item-code,
  label        Label,
  quantity     Quantity,
  price        Cents }

Item-code ::= NumericString (SIZE (7))

Label ::= PrintableString (SIZE (1..30))

Quantity ::= CHOICE { unites      INTEGER,
                      millimeters INTEGER,
                      milligrams  INTEGER }
```

```
Cents ::= INTEGER

Delivery-report ::= SEQUENCE {
  order-number Order-number,
  delivery     SEQUENCE OF Delivery-line }

Delivery-line ::= SEQUENCE { item     Item-code,
                             quantity Quantity }

END
```

As mentioned at the end of Section 4.4, all the types are included in an automatic tagging environment (clause `AUTOMATIC TAGS`), to spare us the trouble of tagging one by one the alternatives of the `CHOICE` types and the optional components of the `SEQUENCE` types.

4.6 A client-server protocol

As a conclusion for our first steps with ASN.1, we specify a data exchange protocol between the sales outlets and the warehouse. The protocol should provide the following functionnalities:

1. make an order and receive a delivery report;
2. consult an item using its code and obtain the available quantity in store;
3. receive the state of an on-going order with the list and the quantity of items delivered so far (order follow-up).

We see that the term '*protocol*' embraces all at once the data to be transmitted, the rules that indicate when a given data should be transmitted and the nature of the service that conveys the data.

Figure 4.1 on the next page shows two systems that cooperate by exchanging messages of the same type called *Application Protocol Data Unit* (APDU, more generally referred to as PDU).

The questions the sale outlets may ask the warehouse and those the latter may ask the former constitute as many alternatives for this PDU, which should be of type `CHOICE`.

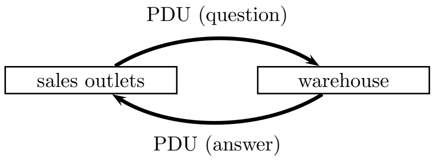


Figure 4.1: Data exchange between two systems: the question and the answer are of the same type

Since we re-use certain types of the previous module called `Module-order`, we now import them in this second module thanks to the `IMPORTS` clause:

```
Protocol DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
IMPORTS Order, Delivery-report, Item-code, Quantity,
        Order-number FROM Module-order ;

PDU ::= CHOICE { question CHOICE {
                   question1 Order,
                   question2 Item-code,
                   question3 Order-number,
                   ... },
                 answer  CHOICE {
                   answer1   Delivery-report,
                   answer2   Quantity,
                   answer3   Delivery-report,
                   ... }}
END
```

Note that `answer1` and `answer3` do not include the order number and that `answer2` does not recall the item code sent in `question2`: we make the implicit assumption that the operations are synchronous, i.e. a question is immediately followed by the corresponding answer. The ROSE protocol ensures such a property. As a result, we consider that information of this nature is useless in the answer.

The two ellipses "...", called *extension markers*, can indicate that should the MicromegAS society require it, we have the possibility of adding new questions and new answers in the next versions of the protocol specification. In this case, the data transfer can carry on even if some sales outlets have kept old versions of the protocols although the central warehouse took over the new version.

We did not mean to tackle the MicromegAS exhaustively in this case study, but a complete and real use of the ROSE protocol (described on page 80) should lead to the definition of three information objects of the class `OPERATION` (see Chapter 15); the first corresponds to sending an order, the second to consulting an item and the third to interrogating an on-going order. Each of these objects would include the ASN.1 type of the question (the argument of the operation) and that of the error list which may be returned in case of failure as shown in the following example:

```
question1 OPERATION ::= {
  ARGUMENT    Order
  RESULT      Delivery-report
  ERRORS      {unavailable | order-rejected}
  SYNCHRONOUS TRUE
  CODE        local:1 }
```

In order to execute one of these three operations, we would have to transmit its `CODE` and its arguments as values of the `Invoke` type of ROSE. The client-server model would then be obtained with no more effort using the numerous functionnalities of the already existing generic ROSE protocol.

4.7 Communicating applications

Now that the ASN.1 specification is written, what should the programmer do in practice (since the programmer and the specifier are hardly ever the same person) to implement the communicating application of the central warehouse or the sales outlets?

The person in charge of this implementation collects all the ASN.1 modules and give them to an ASN.1 compiler. This tool checks whether the specification is properly written, i.e. if it respects the spelling conventions of the language but also its 'rules'. If the specification complies with all the criterions of the ASN.1 standard, the compiler generates (generally) two files: the first for the translation of the ASN.1 module in data types of the *C* language (or some other), the second gathers the *C* functions to code and decode the data for these types.

Using another compiler (a *C* compiler if the language mentioned above is *C*), the two files are 'linked' with the kernel of the communicating application. It is the kernel that for example checks, in the MicromegAS's database whether the items that are being ordered are actually in store, computes the total amount of the order, and edits and prints out the invoice. The *C* compiler generates a binary file that can be executed on the communicating system to transmit the data (send questions and receive answers) that respect the `PDU` type specified in the ASN.1 `Protocol` module.

We see that once the ASN.1 specification has been written, the process that consists in deriving a communication program from it is easy and fast. In particular, if they use an ASN.1 compiler the programmers can ignore the way the data are coded in bit or byte streams for transmission. We shall come back to these notions in Chapter 22.

Chapter 5

Basics of ASN.1

Contents

It is not the language that make the concepts but the concepts that makes the language. And the language used for expressing them always betrays them in one way or another.

Jacques Maritain, *The Peasant in Garonne.*

During our first steps with ASN.1, we mainly defined types (sometimes constrained) that were collected in different modules. We describe in this chapter many other entities that can be handled by ASN.1. Every concept is not described into detail here (a thorough description can be found in the second part 'User's Guide and Reference Manual', on page 95), this chapter is meant to provide an overview of the various possibilities offered by the notation and it shows how these functionalities match with one another to introduce our next chapter.

Before describing these concepts, we start with general spelling conventions to be respected in ASN.1.

5.1 Some lexico-syntactic rules

All the identifiers, references and keywords begin with a letter followed by another that can be upper-case or lower-case, by a digit or a dash "-"[1]. These are some valid ASN.1 identifiers and keywords:

```
INTEGER
v1515
No-final-dash
MY-CLASS
```

and some others, lexically *invalid*:

```
Final-dash-
double--dash
under_score
1515
3M
```

In addition to this rule, the ASN.1 keywords are spelt exclusively in upper-case letters, except some character string types (such as `PrintableString`, `UTF8String`...) because they are in a way re-named after the primitive `OCTET STRING` type.

The strings can be of three kinds:

1. character strings in quotation marks:

   ```
   "This is a string"
   ```

2. binary strings in quotes followed by the upper-case letter `B`:

   ```
   '01101'B
   ```

3. hexadecimal strings in quotes followed by the upper-case letter `H`:

   ```
   '0123456789ABCDEF'H
   ```

For the numbers, the decimal form (with a dot) is not allowed so that it is impossible to use floats the way they are in other languages (the 'real numbers' in ASN.1 are formally defined with three integers: the mantissa, the base, and the exponent).

The comments start with a double dash "--" and stop either at the end-of-line or at the next double dash if found before. Comments are added to the specification to precise the interpretation associated with a

[1]Note that the underscore is forbidden in identifiers and keywords (see footnote 1 on page 100). This symbol can only appear within character strings nested in quotation marks or in comments.

given part or to describe informally what cannot be done formally with the language.

As in most computing languages, spaces, tabulations, newlines and comments are not interpreted in the ASN.1 specification; they are useless except of course for clarity's sake to separate the lexemes, such as keywords, identifiers, numbers or strings. However, the definition (or "assignment" as in the standard) symbol "`::=`" should not include separators, otherwise it would not be interpreted properly by an ASN.1 tool.

Finally, we shall see that ASN.1 uses the different cases (lower-case/upper-case) to distinguish between the various sorts of entities. Thus, words that begin with a lower-case letter (the *identifiers* and the *value references*) make reading and understanding of the specification easier but do not influence the generation of the bits or bytes to be transmitted.

5.2 Types

The main concept in ASN.1 is that of type. A *type* is a non-empty set of values, which can be encoded to be transmitted. Although comparable with the formalisms available in *Pascal* or *Ada*, for instance, for defining complex data types, ASN.1 types should be specific to this transfer task and provide the adequate functionnalities (like the `OPTIONAL` clause of the `SEQUENCE` or `SET` types for example). Types, such as `BIT STRING` or `EMBEDDED PDV`, are even more specific to telecommunications.

The main ASN.1 basic types are presented in Table 5.1 on the following page. More complex types can be built using these basic types combined by the constructed types presented in Table 5.2 on the next page.

When a type is defined, it should be given a name to reference it in another type assignment. This type reference should begin with an upper-case letter and respect the rules of the previous section. All ASN.1 assignments use the symbol "`::=`"; so does the type assignment:

```
Married ::= BOOLEAN
Age ::= INTEGER
Picture ::= BIT STRING
Form ::= SEQUENCE { name                 PrintableString,
                    age                  Age,
                    married              Married,
                    marriage-certificate Picture OPTIONAL }
```

`BOOLEAN`	Logical values `TRUE` and `FALSE`
`NULL`	Includes the single value `NULL`, used for delivery report or some alternatives of the `CHOICE` type (particularly for the recursive types)
`INTEGER`	Whole numbers (positive or negative), possibly named
`REAL`	Real numbers represented as floats
`ENUMERATED`	Enumeration of identifiers (state of a machine for instance)
`BIT STRING`	Bit strings
`OCTET STRING`	Byte strings
`OBJECT IDENTIFIER`, `RELATIVE-OID`	Unambiguous identification of an entity registered in a worldwide tree
`EXTERNAL`, `EMBEDDED PDV`	Presentation (6th layer) context switching types
...`String`	Various types of character strings (see Table 11.1 on page 175)
`CHARACTER STRING`	Allows negotiation of a specific alphabet for character strings
`UTCTime`, `GeneralizedTime`	Dates

Table 5.1: Basic types

`CHOICE`	Choice between types
`SEQUENCE`	Ordered structure of values of (generally) different types
`SET`	Non-ordered structure of values of (generally) different types
`SEQUENCE OF`	Ordered structure of values of the same type
`SET OF`	Non-ordered structure of values of the same type

Table 5.2: Constructed types

```
Payment-method ::= CHOICE {
  check       Check-number,
  credit-card SEQUENCE { number      Card-number,
                         expiry-date Date }}
```

Note that ASN.1 assignments do not end with a semicolon contrary to many computing languages.

As we shall shortly illustrate by some examples, each component of a `SEQUENCE` or `SET` structured type, and each alternative of a `CHOICE` type is named by an identifier (word beginning with a lower-case letter). These identifiers improve the specification readability but they are not transmitted between the communicating applications (in fact during the encoding, the components are represented by a different mechanism).

In order to inform the receiving machine of the type of the value it will encounter so that this value could be properly decoded, the sending machine's encoder can associate a *tag* with it. By default the encoder associates a tag called 'universal'. Sometimes, this default case is not enough to remove all ambiguities and it is necessary to indicate explicitly the tags before the component or alternative types for the constructed types. A tag is a number between square brackets before a type as in:

```
Coordinates ::= SET { x [1] INTEGER,
                      y [2] INTEGER,
                      z [3] INTEGER OPTIONAL }
Afters ::= CHOICE { cheese [0] PrintableString,
                    dessert[1] PrintableString }
```

The problem of tagging is investigated more thoroughly in Section 12.1 on page 206.

ASN.1 allows recursive type assignments but we should ensure that it contains at least one finite value (i.e. non recursive) because encoding rules do not manage infinite values. However, most of the components of the structured types end with simple types.

Every now and then, we need to restrict the set of values (very often infinite) taken by a type to make the specification more precise (to refine it): it ensures interworking but also improves the encoding (particularly with the packed encoding rules). For doing so, we use subtype constraints, which (almost systematically) appear in round brackets after the type:

```
Lottery-number ::= INTEGER (1..49)
Lottery-draw ::= SEQUENCE SIZE (6) OF Lottery-number
```

```
Upper-case-words ::= IA5String (FROM ("A".."Z"))
Phone-number ::= NumericString (FROM ("0".."9"))(SIZE (10))
Coordinates-in-plan ::=
  Coordinates (WITH COMPONENTS {..., z ABSENT})
```

As a constrained type (or subtype) is also a type, it can be used wherever a type may appear.

Finally, when a new specification is produced where new components are added in a SET, SEQUENCE or CHOICE type or a subtype constraint is extended, two communicating machines, especially in an open network do not always use encoders and decoders produced from the same specification version. In order to prevent a machine from stopping when too much (or too little) data are received, it is possible to denote the place where other types can be expected with the extension marker "...":

```
Type ::= SEQUENCE { component1 INTEGER,
                    component2 BOOLEAN,     -- version 1
                    ... }
```

A second version of this type may then be written:

```
Type ::= SEQUENCE { component1 INTEGER,
                    component2 BOOLEAN,
                    ...,
                    [[ component3 REAL ]],  -- version 2
                    ... }
```

Every new group added to a type assignment can be nested double square brackets "[[" and "]]".

5.3 Values

When defining new types, value members of these types can be defined explicitly. A value assignment also uses the symbol "::=", but a value should be referenced by a word starting with a lower-case letter and must be governed by a type (a name starting with an upper-case letter):

```
counter Lottery-number ::= 45
sextuple Lottery-draw ::= { 7, 12, 23, 31, 33, 41 }
pair Coordinates ::= { x 5, y -3 }
choice Afters ::= dessert:"profiterolles"
```

We should stress once again on the fact that (abstract) values defined in an ASN.1 module are never encoded. Indeed, the values to be transmitted are provided dynamically to the encoder by the computing application. References to ASN.1 values are actually used to improve the readability of the subtype constraints, the default values of some structured type components and the definition of value sets and information objects:

```
upper-bound INTEGER ::= 12
Interval ::= INTEGER (0..upper-bound)
default-value Interval ::= 0
Pair ::= SEQUENCE {
  first  [0] Interval DEFAULT default-value,
  second [1] Interval DEFAULT default-value }
```

5.4 Information object classes and information objects

It is sometimes necessary to express more formally than in comments, the semantic links that exist between types and values defined in a specification. We sometimes want to represent the fact that the component of a structured type depends on the value associated with this structured type.

Semantic links are formalized with information object classes. For the class assignment, the symbol "`::=`" must be used and followed by the keyword `CLASS`. The name of this class should be spelt in upper-case letters. Thus the class:

```
OPERATION ::= CLASS {
  &number              INTEGER UNIQUE,
  &Argument-type,
  &Return-result-type }
WITH SYNTAX { OPERATION NUMBER &number
       TAKES AN ARGUMENT OF TYPE &Argument-type
       AND RETURNS A VALUE OF TYPE &Return-result-type }
```

describes specific operations identified with a unique number, associating every one of them with its argument type and its result type. We do not indicate what this operation does since it does not fall not within the competence of ASN.1 but we specify the data types exchanged when a machine asks another to execute these operations. The field `&number` begins with a lower-case letter and is followed by a type; this field, a value of type `INTEGER`, is used as an identification of the information object because it is followed by the keyword `UNIQUE`. The fields `&Argument-type`

and `&Return-result-type` start with an upper-case letter (but are followed by nothing else); they reference any ASN.1 type (no ASN.1 type could have such a representational potential since it would then be impossible to encode). The block `WITH SYNTAX` defines a more user-friendly syntax to denote the objects of this class.

Every operation is defined as an information object of class `OPERATION`. As in every ASN.1 assignment, the object assignment uses a symbol "`::=`" after the object name (which starts with a lower-case letter) and the class name (entirely in upper-case letters):

```
plan-projection OPERATION ::= {
   OPERATION NUMBER 12
   TAKES AN ARGUMENT OF TYPE Coordinates
   AND RETURNS A VALUE OF TYPE Coordinates-in-plan }
```

As the same specification is known by two communicating machines, the same objects can be shared between these two machines.

All the objects of a given class defined in a specification can be gathered in an object set:

```
SupportedOperations OPERATION ::=
  { plan-projection | translation | symmetry, ... }
```

Note that, contrary to an object, an object set name starts with an upper-case letter. The symbol "`...`" indicates that new objects may *dynamically* be added to this set by the communicating applications.

Information objects are never encoded; the only way of transmitting items of their information is to reference them within a type. For this we use the information object set to constrain the components of a structured type:

```
Execute-operation ::= SEQUENCE {
  code     OPERATION.&number({SupportedOperations}),
  argument OPERATION.&Argument-type
              ({SupportedOperations}{@code}) }
Receive-result ::= SEQUENCE {
  code   OPERATION.&number({SupportedOperations}),
  result OPERATION.&Return-result-type
              ({SupportedOperations}{@code}) }
```

These obscure definitions mean that when an operation, whose number is in the `code` component, should be executed, the operations arguments to be transferred should conform to the type of the field

`&Argument-type` of this object, which denotes this operation. This link (or *table constraint*) '`@code`' indicates that this argument type depends on the operation to be executed. Likewise, if a machine that has executed the operation has to transmit the results, its type should be conditionned by the operation code.

5.5 Modules and specification

Having introduced the various concepts that can be defined and managed with the ASN.1 notation, we now have to explain how to collect them to make up the specification of data transfer in a telecommunication protocol. A specification consists of one or several ASN.1 modules where each module gathers types (mainly), values, information object classes, information objects and information object sets.

A module name begins with an upper-case letter. It can be referenced by a kind of 'universal pointer', called object identifier, in curly brackets after its name.

Here is an example of such a module:

```
Module2 { iso member-body(2) f(250) type-org(1) ft(16)
          asn1-book(9) chapter5(0) module2(1) }
DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
EXPORTS Type2;
IMPORTS Type1, value FROM Module1 {iso member-body(2)
          f(250) type-org(1) ft(16) asn1-book(9)
          chapter5(0) module1(0)};

Type2 ::= SEQUENCE OF Choice
Choice ::= CHOICE { a INTEGER (0..value),
                    b Type1 }
END
```

The clause `AUTOMATIC TAGS` indicates that the specifier needs not care of tagging (in square brackets "`[`" and "`]`") the components of the structured types that are defined afterwards in the module and leaves it to the compiler.

The clauses `IMPORTS` and `EXPORTS` (not mandatory) define the module interface. The `IMPORTS` clause lists the assignment names defined in other modules and imported in the current module (i.e. those which will be used within the current module as if they were defined there). The

`EXPORTS` clause list the assignment names that should be accessible outside the current module and therefore could be imported within another module.

As said in Chapter 4 for our first steps with ASN.1, the specifier is free to order the assignments of a module as judged fit (even though a top-down approach is generally adopted, starting from the most general to end in with the basic types) provided that each name is used only once (in the same upper/lower-case) to reference a single assignment. Moreover, the specification can be splitted into as many modules as necessary for clarity, comprehension, and re-usability.

As a conclusion we can recall that all the assignments (or definitions) that can appear in an ASN.1 specification are only meant to be linked in one way or another with the data to be transferred (i.e. with the data types), which are the *raison d'être* of ASN.1. In the rest of this text, we shall regularly come back to this point. In fact, as seen in this chapter, some ASN.1 features are specific to data transfer; they do not appear in any other abstract notations or computing languages.

Chapter 6

History

Contents

"Dear Sirs, before banishing Greek, have you even thought about it? After all what would *you* do without Greek? Would Chinese or a Coptic or Syrian Bible make you turn away from Homer and Plato?"

Paul-Louis Courier, *Open Letter to the Members of the* Académie des Inscriptions et Belles-Lettres.

The reference to standardization becomes a common place as issues and conflicts get more obvious. Standardization consists of a collection of specifications, the standards, which are available to all the key players of a technico-economic sector.

A standard is defined as "*a documented agreement containing technical specifications or other precise criteria to be used consistently as guidelines, or definitions of characteristics, to ensure that materials, products, processes, and services are fit for their purpose*".

Now that the fundamentals of ASN.1 have been exposed we can describe in detail how the notation has evolved since its creation in the early eighties. But before telling this story, we should present first the two main organizations for standardization: ISO and ITU.

6.1 International Organization for Standardization (ISO)

The International Organization for Standardization (ISO)[1] was established by the United Nations in 1946. It was originally in charge of setting up international standards in numerous areas except electricity, electronics and electro-technics, which fell within the competence of the International Electrotechnical Commission (IEC).

It gathers about a hundred countries that delegate their own standardization committee (called *National Body* or NB). The USA is represented by the *American National Standard Institute* (ANSI)[2], France by the *Association Française de NORmalisation* (AFNOR), Great Britain by the *British Standards Institute* (BSI)[3], Germany by the *Deutsches Institut für Normung* (DIN)[4] and Japan by the *Japanese Industrial Standards Committee* (JISC)[5], to mention but a few. Other organizations may join the discussions and propose contributions, but they cannot vote.

[1]ISO is not really an acronym; it is the Greek prefix that means 'equal' and is shared among the three official languages of this organization: English, French and Russian. The server http://www.iso.ch provides, among other things, references to all the standards as well as the list of the committee members. An introduction to ISO can be consulted at http://www.iso.ch/infoe/intro.htm.

[2]http://www.ansi.org

[3]http://www.bsi.org.uk

[4]http://www.din.de

[5]http://www.jisc.org/eorg1.htm

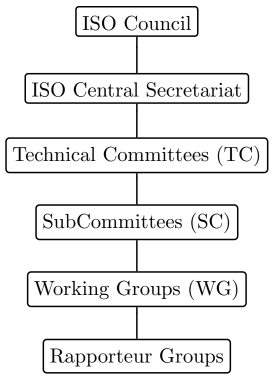


Figure 6.1: ISO breakdown structure

As seen in Figure 6.1 ISO is divided in 172 *Technical Committees* or TC in charge of various standardization domains. The subjects are shared among the *SubCommittees* (SC), also divided in *Working Groups* (WG).

Until 1987, the OSI standards depended on TC 97, called "Telecommunications and Information Exchange Between Systems". In 1987, ISO and IEC agreed that they were both concerned with Information Technology and formed a joint technical committee called JTC 1[6]. Figure 6.2 on the following page gives only a partial viewpoint of the structure that focuses on the issues to be discussed later. The JTC 1 Secretariat is ensured by ANSI.

Designing an international standard is often a tedious and complex process[7]. A WG gathers experts in charge of a preliminary study which, if an agreement has been found, leads to the publication of a *Draft Proposal* (DP). This drast remains under discussion for twelve weeks (it can be voted 'Yes', 'Yes with comments' or 'No with comments'). Once the comments have been taken into account so that the number of negative votes could be reduced, the document is submitted to the TC that adopts it as a *Draft International Standard* (DIS) — if only a few negative votes remain — and sends it again to go round the members for twelve weeks under the same voting conditions as before.

[6]http://www.jtc1.org (today, JTC 2 is still unknown!). A complete description can be found at http://www.iso.ch/meme/JTC1.html. JTC 1 is described thoroughly at http://www.iso.ch/dire/jtc1/directives.html.

[7]The process is detailed at http://www.iso.ch/infoe/proc.html.

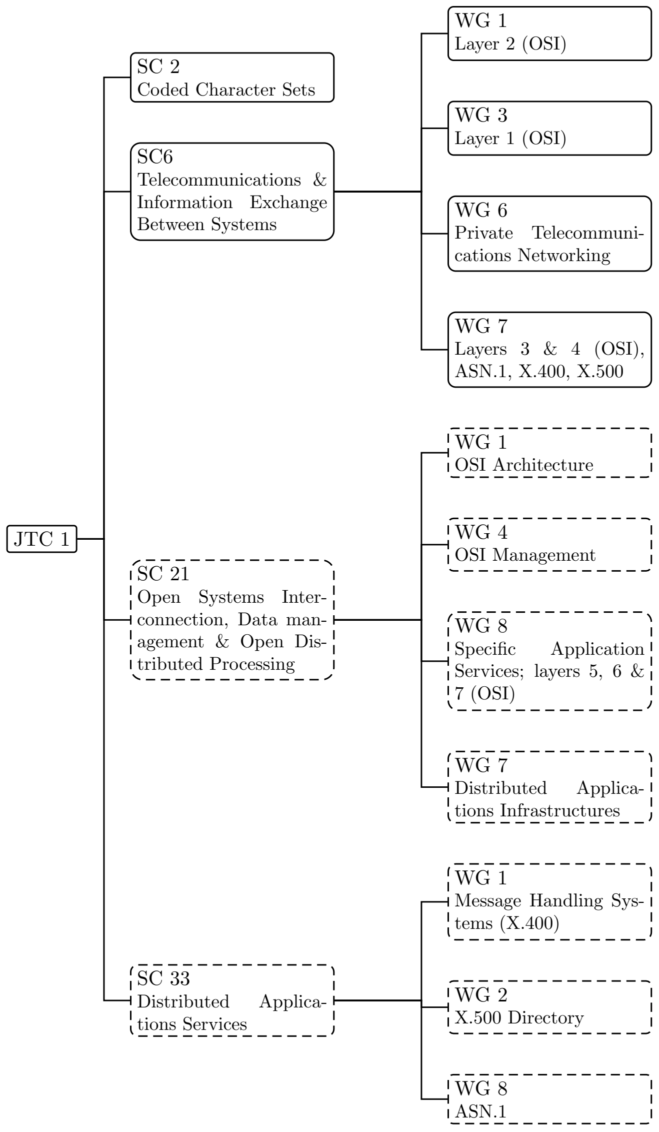

JTC 1
SC 2
Coded Character Sets
SC6
Telecommunications & Information Exchange Between Systems
WG 1
Layer 2 (OSI)
WG 3
Layer 1 (OSI)
WG 6
Private Telecommunications Networking
WG 7
Layers 3 & 4 (OSI), ASN.1, X.400, X.500
SC 21
Open Systems Interconnection, Data management & Open Distributed Processing
WG 1
OSI Architecture
WG 4
OSI Management
WG 8
Specific Application Services; layers 5, 6 & 7 (OSI)
WG 7
Distributed Applications Infrastructures
SC 33
Distributed Applications Services
WG 1
Message Handling Systems (X.400)
WG 2
X.500 Directory
WG 8
ASN.1

Figure 6.2: Partial description of JTC 1 structure (groups and committees surrounded by a dashed line have been dismantled)

When approved by three quarters of the committee, the document is submitted to the Central Secretariat (see Figure 6.1 on page 55) to be translated into French and Russian (the other two official languages beside English). The document is then 'granted' the status of *International Standard* (IS). It is finally submitted to the ISO Council and is published by the committee members.

As far as the ASN.1 standard is concerned, its development is usually carried out by an ISO group called Rapporteur Group and the document produced should be approved before being sent to ITU-T for official ratification in a plenary meeting. However, as described in Section 6.3.1 on page 60, for historical reasons, ASN.1 stemmed from CCITT works, which influenced it by numerous contributions. In the eighties, a collaboration did exist between the two committees before ISO became the main actor around 1990. But things can change...

All along this process, an absolute consensus on every issue, or at least substantial, is ensured by all the members and structures involved (since it often requires to modify the text of the standard in question). The member committees are under no obligation to adopt the standards and their contributions should be understood to be on a voluntary basis.

The addition of a new functionality to a standard is done through a *Proposed Draft Amendment* (PDAM) which, after reviewing during meetings and circulation among the member committees for approval (vote), reaches the status of *Draft Amendment* (DAM) and finally *amendment*. When relatively minor errors or misunderstandings are spotted in a standard, another procedure, faster than the amendments, can bring them to public attention: the *defect reports*, which once approved turn into *Technical Corrigenda* that are published in parallel. The number of Amendments and Technical Corrigenda can be important and it is sometimes tedious to get all of them: in order to make life easier for the users of a potentially amended standard, it is preferable that the corresponding Rapporteur Group issues a new edition of its standard including all these changes.

The organization of ISO, the document levels and evolution from one to the other is thoroughly described in [Lar96, Chapter 1] and [Hor96].

6.2 International Telecommunications Union (ITU)

The International Telecommunications Union (ITU) whose headquarters are based in Geneva, became a specialized institution of the United Nations in 1947 (it was preceded by the International Telegraph Union founded in 1865). It gathers 188 countries represented by their public telecommunication operator and around 450 other members from the private sector called *Registered Private Operating Authorities* (RPOA), such as *AT&T* and *Bell Telephone*. The operators from any other organizations are accepted but have not the right to vote.

The objectives of the ITU is "*to promote and to offer technical assistance to developing countries in the field of telecommunications, and also to promote the mobilization of the material and financial resources needed for implementation*", and "*to promote the extension of the benefits of the new telecommunication technologies to all the world's inhabitants*".

The ITU is made of five permanents organs among which is the Consultative Committee on International Telephony and Telegraphy (CCITT) in charge of telecommunication networks, as well as wired transmission of voice, data and television. After the re-organization of ITU in 1992, the CCITT became ITU-T (ITU - Telecommunication Standardization Sector).

The ITU-T publishes recommendations referenced by a letter (one for each domain) and a number. The sectors we are more particularly interested in are:

- F: Non-Telephone Telecommunications Services;
- H: Audiovisual and Multimedia Systems;
- Q: Switching and Signalling;
- T: Terminals for Telematic Services;
- V: Data Communications Over the Telephone Network;
- X: Data Networks and Open System Communications;
- Z: Programming Languages.

Until 1992, all of the CCITT recommendations approved by the committee members in plenary meetings were published every four years (even if some parts had not been completed yet)[8] as a leaflet with a different colour for each release: yellow for 1976-80, red for 1980-84, blue[9] for 1984-88 and white for 1988-92.

ITU-T now publishes each standard separately as soon as it is considered as stable. Besides, since 1994, ISO/IEC and ITU-T jointly publish texts with one reference for ISO/IEC and another for ITU-T, in order to prevent any inconsistencies between them as it was the case for ASN.1[10].

The standard can be purchased from the ITU-T secretary in Geneva or on their web site[11]. The ITU-T members that want to provide international services in domains of the organization's competence tend to apply these recommendations.

The ITU-T is divided into *Study Group* (SG) where we find:

- SG VII, "Data Networks and Open System Communications", responsible, among other things, for interconnection (hence ASN.1), for the X.25 network standard, for the X.400 directory standard for e-mail, for security and for network management;
- SG VIII, "Terminals for Telematic Services", in charge of Teletex, of Videotex and of the character sets.

The work of each SG is subdivided into 'questions': some of them can be very general such as Q24/7, which concerns the OSI model and some others more precise like updating an already published recommendation. Every question of this work schedule ends in the release of a new recommendation or the amendment of an existing one.

ITU-T is more thoroughly described in [Hor96].

[8]Since ISO publish their documents when the studies are over, it sometimes implied correcting mistakes from the CCITT! However, CCITT policy is well-founded: the marketing and investment programs of its members (the networks operators) cannot wait for the mistakes of a standard to be corrected.

[9]Hence the famous 'Blue Book' in which, for the first time, appeared the word 'ASN.1' whereas the first version of the notation belongs to the red book.

[10]We can mention another example of inconsistency for the X.400 e-mail standard series: in the CCITT text, the e-mails could only cross a border through a public operator or RPOA, which was no restriction since every member belonged to either of the two categories. On the other hand, the ISO text allows a private system to be connected across a border.

[11]http://www.itu.int/itudoc/itu-t/rec.html

6.3 The great story of ASN.1

This amazing story begins in the summer of 1982. Many people who worked on the development of standards on the Application Layer had spotted the same problem: the data structures had become too complex to allow bespoke procedures for encoding and decoding in bits or bytes (the word 'Open', term of utter importance in OSI, had appeared in 1978). Likewise, as the compilers took over the assemblers, the common belief was that encoders should be generated automatically from a specification (to make the latter the equivalent to a computer program).

6.3.1 Birth

James White [Whi89] and Douglas Steedman (*Bell-Northern Research*, Canada) independently proposed the basis of a notation and an algorithm that could define the format of the encoding bits for the e-mail Message Handling Systems (MHS) protocols. This notation and encoding scheme were machine-independent and could convey complex data structures.

James White was at the time the editor of the X.400 (MHS) recommendation series at CCITT and designed for *Xerox Corporation*, the *Courier*[12] notation [Xer81] that could represent data to be transferred by Remote Procedure Call (RPC) from the Xerox Network Services (XNS) protocol series. *Courier* was the first external data notation to be well-known; it also positively influenced the XDR from *Sun Microsystems Inc.* and the *NDR* from *Apollo Computer Inc.*: all this is discussed in greater length in Chapter 24. It is therefore an adaptation of *Courier* notation that James White proposed.

After meeting Douglas Steedman, the consensus was found on James's position (for the benefit of future users as acknowledged by D. Steedman himself in his book [Ste93]!).

In 1984, CCITT standardizes the notation under the reference X.409 ('Red Book'). The X.409 recommendation called 'Message Handing Systems: Presentation Transfer Syntax And Notation', had 32 pages in its

[12]The syntax of the *Courier* notation was adapted from *Xerox*'s *Mesa* used for developing systems composed of separate modules with a controlled information sharing. Reading the *Mesa* manual [MMS79] is quite singular for someone who is familiar with the language: we can recognize obviously some syntactic features of ASN.1 but also notice that a semicolon was needed at the end of a type definition, or that a plethora of square brackets has now been replaced by (a plethora of) curly brackets!

English Version. Its purpose and scope were to define "*the presentation transfer syntax used by application layer protocols in message handling systems and by the document interchange protocol for the telematic services. In the architecture of open systems interconnection (OSI), a presentation transfer syntax is used to represent information exchanged between application entities*".

It defines the built-in types `ANY`, `BIT STRING`, `BOOLEAN`, `CHOICE`, `INTEGER`, `NULL`, `OCTET STRING`, `SEQUENCE`, `SEQUENCE OF`, `SET` and `SET OF`, as well as the character string types `IA5String`, `NumericString`, `PrintableString`, `T61String`, `VideotexString`, and the time types `GeneralizedTime` and `UTCTime`: 14 tags of the `UNIVERSAL` class altogether, but also (already!) the unfortunate macros (see Chapter 16). The presentation of each type was followed by its value notation and its encoding called at the time 'standard representation of type'. Every form of encoding (primitive or constructed, of fixed or unknown length) was already defined. The grammar presented in a condensed format, with all the alternatives of a rule on the same line, took only slightly more than a page!

The X.409 notation was totally independent of the MHS system, partly because the objects handled by the e-mail protocol (no size limit, several string types, linked structures, numerous options) were arbitrarily complex[13]. As a result, many groups that were working on standardization of OSI applications realized it could also prove useful to them.

6.3.2 Baptism

Initiated by John Larmouth[14] (Salford University, UK), the X.409 notation was adopted by the 'OSI world' almost immediately. He distinguishes abstract notation from transfer syntax in two different documents and proposes the term 'ASN.1'[15] whereby ISO implies by the number '`1`' that several other notations could be standardized after that (such a thing never occurred so far, and the unlikely ASN.2 would have

[13]As shown in Figure 7.1 on page 82, an e-mail is made of an envelop and a content; the envelop includes a list of addressees, the sending date, etc.; every addressee is also described in a structure and so on.

[14]http://www.salford.ac.uk/iti/jl/larmouth.html

[15]The dot "." is history: at the time of teletype terminals, the number '`1`' and the upper-case '`I`' had the same graphical representation and typing mistakes may have induced a confusion between `ASN1` and `ANSI` (the American standardization organization).

to bring substantial features to claim the name). When ASN.1 was first published, so much more readable was the text that some people called it 'the English version of X.409'!

Though technically equivalent to X.409, the two ISO documents were completely rewritten because of the emergence of the concept of Presentation layer. It had also become necessary to distinguish a piece of information on the Application layer from its representation, thereby allowing standardization and use of several abstract transfer syntaxes[16] (those concepts were accepted much later by CCITT).

Though reluctant at first, CCITT agreed in 1985 to work jointly with ISO and adopted the structure of two distinct documents as well as the terms 'ASN.1' and 'BER'. The two documents were published in 1987 under the references ISO 8824 and ISO 8825. Compared to those of CCITT, they included three new character string types together with the notion of registration tree and the `OBJECT IDENTIFIER` type.

6.3.3 The 1989 and 1990 editions

In 1987, as described in Section 6.1 on page 54, ISO merged its activity in information technology domains with that of IEC and created the joint technical committee JTC 1, which took over the work on ASN.1.

In 1989, CCITT published two documents referenced by X.208 (ASN.1) and X.209 (BER), which replaced the X.409 recommendation. New features resulting from common work with JTC1 were introduced: subtypes, floats (`REAL` type), pointers (`ANY DEFINED BY` type) and the default tagging modes (`IMPLICIT TAGS` and `EXPLICIT TAGS`). Their publication in the X.200 series, called 'General OSI Infrastructure', acknowledged ASN.1 universally as a specification language for the Application layer.

In 1990, ISO published new releases for the ISO 8824 and ISO 8825 standards. Compared to X.208:1988[17], there are three minor differences:

- although X.208:1988 imposes the use of an identifier of type `INTEGER`, `OBJECT IDENTIFIER` or `ENUMERATED` after the clause `ANY`

[16]This term leads back to the term 'Basic' in BER (Basic Encoding Rules), that gives way to the standardization of many other encoding rules as described in Chapter 18.

[17]We use the symbol ":" as a convention for denoting a standard with respect to its publication date.

`DEFINED BY`, ISO 8824:1990 accepts a `CHOICE` between `INTEGER` and `OBJECT IDENTIFIER` (see Section 12.8 on page 241);

- the lexical token localvaluereference in the macro notation starts with a lower-case letter in X.208:1988 but with an upper-case letter in ISO 8824:1990 (see Chapter 16);

- the technical corrigendum (whose final edition is that of April 1991), which introduces the symbol ":" in the notation of a value of type `CHOICE` (see Section 12.6 on page 235) or `ANY` (see Section 12.8 on page 241) is not taken into account by X.208:1988; this technical corrigendum makes the updating of every existing specification necessary. This, however, could not be prevented since, without changing the ASN.1 grammar, parsing specifications would have been impossible.

Since 1998, ISO and ITU-T have jointly conceived and published their documents, thereby avoiding the tricky problem of consistency mentioned above.

6.3.4 The 1994 edition

A new edition of the standard called ASN.1:1994 (1994 is the year of the final draft but not its publication year) was adopted by voting at ISO in late 1995. Due to numerous additions and the introduction of new concepts, it was divided into 4 parts:

- ITU-T Rec. X.680 (1994) | ISO/IEC 8824-1:1995[18]: Specification of Basic Notation,

- ITU-T Rec. X.681 (1994) | ISO/IEC 8824-2:1995: Information Object Specification,

- ITU-T Rec. X.682 (1994) | ISO/IEC 8824-3:1995: Constraint Specification,

- ITU-T Rec. X.683 (1994) | ISO/IEC 8824-4:1995: Parameterization of ASN.1 Specifications,

[18]The publication years are different between ISO and ITU-T, but the two documents are similar.

that would, a few months later lead to two amendments and one technical corrigendum:

- ITU-T Rec. X.680/Amd.1 (1995) | ISO/IEC 8824-1:1995/Amd.1: Rules of Extensibility,
- ITU-T Rec. X.680/Corr.1 (1995) | ISO/IEC 8824-1:1995/Corr.1: Technical Corrigendum 1,
- ITU-T Rec. X.681/Amd.1 (1995) | ISO/IEC 8824-2:1995/Amd.1: Rules of Extensibility.

The main differences and new concepts are the following:

- the identifiers `mantissa`, `base` and `exponent` appear in the definition of `REAL` values;
- the identifiers of `SET` or `SEQUENCE` components and those of `CHOICE` alternatives become mandatory to avoid ambiguities when defining values of these types;
- types can be tagged automatically with the clause `AUTOMATIC TAGS` inserted in the header of the module;
- the identifiers of an `ENUMERATED` type can be numbered automatically;
- the constructed types (`SEQUENCE`, `SET`, `CHOICE`) and some subtype constraints can be declared as extensible using the "..." symbol (called extension marker), which enables communication between two machines that do not share the same protocol version (see Section 12.9 on page 244): the machine that receives too much information or not enough compared to what was expected can manage the difference:

```
ExtensibleType ::= SEQUENCE { component1 Type1,
                              ... }
ExtensibleConstraint ::= INTEGER (1..5, ...)
```

- the exception marker "`!`" indicates error codes that should be used by the communicating application when receiving data that do not comply with an extensible type definition (see on page 247):

```
ExtensibleType ::= SEQUENCE { component1 Type1,
                              ...! exc-extended-type }
exc-extended-type INTEGER ::= 3
```

- set combination allows unions (`UNION`), intersections (`INTERSECTION`) and exclusions (`EXCEPT`) of subtype constraints (see Section 13.11 on page 285):

```
ISO-10646-String ::= BMPString (FROM (Level2 ^
      (BasicLatin | HebrewExtended | Hiragana)))
```

- every concept (type, value, information object...) can be parameterized to generalize them with respect to their different uses and avoid equivalent definitions (see Chapter 17);

- new character string types such as `UniversalString` and `BMPString` can encode all the alphabets of the [ISO10646-1] or [Uni96] standards (see Section 11.10 on page 183);

- new presentation context switching types like `EMBEDDED PDV` (a more functional definition of the `EXTERNAL` type, whose definition has also been improved) and `CHARACTER STRING` (for negotiating any character set) are introduced (see Chapter 14);

- macros are removed[19] because they were poorly documented thus badly used and impossible to automatize in all their generality (see Chapter 16); these are replaced by a new concept of information object class whose definition is unambiguous, easier to understand and handier to use[20] (see Chapter 15):

```
MY-ATTRIBUTE ::= CLASS {
  &id           OBJECT IDENTIFIER,
  &Type,
  &other-field  OTHER-CLASS OPTIONAL }
WITH SYNTAX { &id OF TYPE &Type
              [ASSOCIATED WITH &other-field] }
```

[19]In fact, the text is transferred in an annex of the standard to preserve a means of reading and understanding obsolete specifications.

[20]Except a couple of curly brackets, the syntax of macro instances is in general exactly the same as that of their equivalent information objects.

these information object classes and information objects constitute the main contribution of the 1994 edition;

- constraints on information object class fields are introduced; they allow in particular to represent the `ANY DEFINED BY` type that could not be automatically taken into account by a compiler even when a table describing the associations was included in comments within the module (see Section 12.8 on page 241):

  ```
  MyAttributes MY-ATTRIBUTE ::=
    { attrib1 | attrib2 | attrib3 }
  MyAttribute ::= SEQUENCE {
    &type  MY-ATTRIBUTE.&id({My-Attributes}),
    &value MY-ATTRIBUTE.&Type({My-Attributes}{@type}) }
  ```

- two frequently used information object classes (`TYPE-IDENTIFIER` and `ABSTRACT-SYNTAX`) are standardized together with the `INSTANCE OF` type.

The encoding rules standard is divided into two parts (and a technical corrigendum!):

- ITU-T Rec. X.690 (1994) | ISO/IEC 8825-1:1995: ASN.1 Encoding Rules: Specification of Basic Encoding Rules, Canonical Encoding Rules, and Distinguished Encoding Rules,

- ITU-T Rec. X.690/Corr.1 (1995) | ISO/IEC 8825-1:1995: Technical Corrigendum 1,

- ITU-T Rec. X.691 (1995) | ISO/IEC 8825-2:1995: ASN.1 Encoding Rules: Specification of Packed Encoding Rules,

which officialize the canonical encoding rules (necessary for encrypted-data transmission for example), the distinguished encoding rules (used mainly by the X.500 directory) and the packed encoding rules (used more and more for multimedia data transfer) that are discussed in Part III from page 391 onwards.

In July 1995, John Larmouth proposed a contribution to remove ambiguities from the standard and to start defining compatibility rules between types. The initial goal of the proposal was to clearly define what is semantically correct in ASN.1 to prevent software designers from taking interpretations of the standard. The bottom-line idea was to clarify the meaning to be given to "*should be of the same type as*", which is frequently used in the standard and does not always say whether the types should be tagged, constrained... Unfortunately called 'formal model' at first, its name was changed afterwards for 'semantic model'.

In May 1996, during an ISO meeting at Kansas City, it was decided that:

- only one insertion point should be allowed in extensible types and a second extension marker "`...`" is now permitted (in which case the insertion point is located before the second marker);

- successive levels of extensions must be surrounded by double square brackets "`[[`" and "`]]`":

```
Record ::= SEQUENCE {
  userName              VisibleString,
  password              VisibleString,
  ...,
  [[ lastLoggedIn       GeneralizedTime OPTIONAL,
     minutesLastLoggedIn INTEGER        -- version 1
  ]],
  [[ certificate        Certificate    -- version 2
  ]],
  ... }
```

- the definition of the registration tree and that of its first arcs (`iso`, `itu-t` and `joint-iso-itu-t`) have moved to the [ISO9834-1] standard;

- the semantic model should be rewritten before publication as an amendment;

- the 'millennium bug' for the `UTCTime` type is addressed (see Section 11.17 on page 202);

- the work on global parameters was given up because of lack of interest from the users and the other standardization groups; the idea was originally to avoid fastidious repetitions of parameters frequently used in a given specification (for example, the boundaries of an integer interval);

- the study concerning dynamically constrained types should be pursued[21]: such special parameters used only by PER could be inserted between the symbol "::=" and the type assignment's left-hand side; these parameters' values are transmitted during communication of the encoded value and provide, for instance, an efficient encoding for an array:

```
My-test ::= { OBJECT IDENTIFIER abstract-syntax,
              OBJECT IDENTIFIER transfer-syntax,
              INTEGER           min-length,
              INTEGER           max-length }
  SEQUENCE OF CHARACTER STRING
    (SIZE (min-length.. max-length))
    (WITH COMPONENTS {...,
      identification (WITH COMPONENTS {
        syntaxes ({abstract abstract-syntax,
                   transfer transfer-syntax})})})
```

6.3.5 The 1997 edition

In May 1997, considering the important number of amendments for each part of the ASN.1:1994 standard as well as those still in progress, the working group propose a new version in order to make the specifier's job easier (it always proves difficult to obtain all the numerous amendments from the standardization organizations and one should always be reluctant to manual cut/pastes for updating the standard document since these are prone to error).

Compared to the 1994 version, the new release, called ASN.1:1997, features the following changes:

- the registration tree definition moves to [ISO9834-1];
- the second extension marker and the version double square brackets are inserted in the types `SEQUENCE`, `SET` and `CHOICE`, and a tutorial on extensibility is included in the appendices;
- the `UTF8String` type is defined (see Section 11.12 on page 190);
- the appendices concerning the `ANY` type and the macros have been removed.

[21] See documents ISO/IEC JTC 1/SC 21 N 9735, N 10999, N 11001, N 10423.

After the reorganization of JTC 1, SC 21 was dissolved in late 1997. The standardization activities for ASN.1 joined SC 33 ('Distributed Applications Services') with ODP, ODMA, X.400, X.500, *Enhanced LOTOS*, Transaction Processing, Systems Management and the OSI Application layer protocols. Peter Furniss was appointed OSI maintenance rapporteur[22] in SC 33, which means that he is in charge of the above standards that are in a stable state to avoid developping them in any ISO working group (nevertheless note that in spite of this, ASN.1 still has its own Rapporteur group[23]).

In January 1998, an editing meeting on ASN.1 *semantic model* was organized in Manchester (UK). John Larmouth had completely rewritten his contribution to the model and refined it, removing all that did not deal with type compatibility. Having done this, the group concluded with the results presented in Section 9.4 on page 121.

At the end of the first semester 1998, the (still) new SC 33 was dissolved since no international organization volunteered for the Secretariat. Thus ASN.1 joined the SC 6 ('Telecommunications and Information Exchange Between Systems'), and more specifically WG 7 that dealt with the layers 3 and 4 of the OSI model, with X.400 and with X.500 (see Figure 6.2 on page 56). The ASN.1 Rapporteur group now manage by themselves the defect reports, which are not taken into account in the context of OSI maintenance (see footnote 22).

The sub-committee SC 6 of ISO keep in touch with the ITU-T study group SG VII, which facilitates common work on ASN.1.

In January 1999, a meeting was organized in Lannion (France) for ten days. Among the numerous topics, we can mention:

- the progression of an amendment that would include the semantic model (see Section 9.4 on page 121) as Annex F of the [ISO8824-1] standard;

- the withdrawal of the dynamically constrained types (see on page 67) because their power did not necessarily make up for their

[22] The ASN.1 maintenance web page is at http://www.furniss.co.uk/maint/asn/. It contains all the amendments and defect reports for each part of the standard. Specifiers may find there the most up-to-date (and valid!) documents.

[23] This working group is now open to everyone, provided one's action remains restricted to discussions on the standards and corrections of the errors it may contains (it is not necessary to stand in for a National Body any more).

complexity and despite their undeniable interest for a lot of communicating applications, it seemed difficult to standardize them without prototyping beforehand;

- the creation of the `RELATIVE-OID` type for relative object identifiers (see Section 10.9 on page 167);

- the investigation on introducing the underscore "_" as an equivalent for the dash "-" in references and identifiers to homogenize ASN.1 labelling with that of other formal notations (see footnote 1 on page 100 and Section 23.1 on page 476);

- the correction of some defect reports;

- a proposition for defining a new formalism, called encoding control, associated with ASN.1, which would allow the user to specify other encoding rules or to overload existing standardized encoding rules (like BER, PER...) for the needs of a specific domain or application (see Section 21.6 on page 459).

In July 1999, the working group met in Geneva while an ITU-T SG VII/WP 5 meeting was being held at the same time. The following documents prepared during the Lannion meeting were finalized for approval by ITU-T in July 1999 (their approval by ISO is planned for February 2000):

- the semantic model presented in Section 9.4 on page 121 is subject to three amendments:

  – ITU-T Rec. X.680 (1997)/Amd.2 (1999) | ISO/IEC 8824-1:1998/Amd.2,

  – ITU-T Rec. X.681 (1997)/Amd.1 (1999) | ISO/IEC 8824-2:1998/Amd.1,

  – ITU-T Rec. X.683 (1997)/Amd.1 (1999) | ISO/IEC 8824-4:1998/Amd.1;

- the new `RELATIVE-OID` type of relative object identifiers (see Section 10.9 on page 167) introduces three amendments:

  – ITU-T Rec. X.680 (1997)/Amd.1 (1999) | ISO/IEC 8824-1:1998/Amd.1,

  - ITU-T Rec. X.690 (1997)/Amd.1 (1999) | ISO/IEC 8825-1:1998/Amd.1,
  - ITU-T Rec. X.691 (1997)/Amd.1 (1999) | ISO/IEC 8825-2:1998/Amd.1;

- and four technical corrigenda clarify new points of the standard:
  - ITU-T Rec. X.680 (1997)/Corr.1 (1999) | ISO/IEC 8824-1:1998/Corr.1,
  - ITU-T Rec. X.681 (1997)/Corr.1 (1999) | ISO/IEC 8824-2:1998/Corr.1,
  - ITU-T Rec. X.690 (1997)/Corr.1 (1999) | ISO/IEC 8825-1:1998/Corr.1,
  - ITU-T Rec. X.691 (1997)/Corr.1 (1999) | ISO/IEC 8825-2:1998/Corr.1.

All the material aforementioned has already been taken into account in this text.

In addition, two new work items were created and may lead to new standards in 2001:

- one about the concept of encoding control (see Section 21.6 on page 459);
- another about the encoding rules for XML called XER and the mapping between ASN.1 types and XML Schema datatypes (see Section 21.5 on page 458).

In October and December 1999, the ASN.1 working group held two meetings in Somerset (New Jersey, USA). Three main issues were discussed: supporting XML in ASN.1, encoding control notation and examination of a few defect reports submitted by AFNOR. These defect reports will give rise to some technical corrigenda on the standard sections involved.

In March 2000, a meeting was held at the ITU-T headquarters in Geneva. It mainly focused on the encoding control notation and the production of technical corrigenda among which we can find:

- [ISO8824-1DTC2], which allows the exception marker "`!`" after the extension marker "`...`" in an `ENUMERATED` type and introduces the clause "`EXPORTS ALL;`";

- [ISO8824-1DTC3], which allows the decimal notation to define `REAL` values;
- [ISO8824-1DTC4], which defines a subtype constraint based on regular expressions introduced by the keyword `PATTERN` that can be applied on the character string types (see Section 13.7 on page 271);
- [ISO8824-3DTC2], which defines a subtype constraint introduced by the keywords `CONTAINING` and `ENCODED BY` that can only apply to the types `BIT STRING` and `OCTET STRING` (see Section 13.10 on page 283).

Other ASN.1 meetings have been planned every other month for the year 2000: they will mostly be dedicated to the new ECN notation (see Section 21.6 on page 459).

6.4 Compatibility between the 1990 and 1994/1997 versions

The participants of the July 1997 meeting in London and the ITU-T members agreed that the ASN.1:1990 standard should be removed from ISO and ITU-T proceedings. This question was very much debated among the members of other standardization groups partly because it seemed difficult to remove the edition of a standard that was quoted in a significant number of other standards. Moreover, as some standards are only being maintained without a working group's being dedicated to them, it proves impossible to re-write them completely.

Even if most of the Application Layer standards are being updated to comply with ASN.1:1997, it is very unlikely that all specifiers will adapt the existing modules (and unfortunately many questions related to the `ANY DEFINED BY` type of ASN.1:1990 on various newsgroups on the Internet bear out this hypothesis). As a consequence, the SC 33 decided to maintain the ASN.1:1990 standard while strongly recommending the migration towards ASN.1:1997 as much as possible and advising the designers to use it for all new specifications.

We now explain how modules that have different ASN.1 versions can fit in the same specification and we describe the actions to be undertaken for updating the ASN.1:1990 modules into ASN.1:1994/97.

6.4.1 Composition rules of the two versions

The following rules are extracted from [ISO8824-1, annex A]:

- any module should entirely conform to ASN.1:1990 only or to ASN.1:1994/97 only; the specifier should indicate for each module the version used by means of comments such as:

  ```
  -- version BIT STRING { v1990(0), v1994(1),
  --                      v1997(2) } ::= v1994
  ```

  As a result, if some part of a module needs to be updated, this should be divided into two separate modules.

- A module that conforms to ASN.1:1990 can import ASN.1:1994/97 types or values (only) if these types could as well be written in ASN.1:1990 (for example it is possible to import `SEQUENCE` or `PrintableString` types, but it is impossible for a `UTF8String` type); extensible types cannot be imported; a tool that conforms to ASN.1:1990 is very unlikely to support imports of structured types from a module featuring the `AUTOMATIC TAGS` clause in its header.

- A module written in ASN.1:1994/97 cannot import macros; if `SEQUENCE`, `SET` or `CHOICE` types are imported, value definitions can be written only for types in which all the identifiers appear and for those which do not use `ANY` types; in the same way, the `WITH COMPONENTS` subtype constraint can be used only for components preceded by an identifier.

6.4.2 Migration from ASN.1:1990 to ASN.1:1997

Since the ASN.1:1994 and ASN.1:1997 standards benefit from real improvements compared to the 1990 version, we highly recommend to update 'old' specifications. The following rules are extracted from [ISO8824-1, annex A]:

1. every component of a `SEQUENCE`, `SET` or `CHOICE` type must have an identifier that must be used in the value definitions of these types;

2. the `mantissa`, `base` and `exponent` identifiers must be added to every value of type `REAL` and `base` should be restricted `2` or `10` as much as possible;

3. the `ANY` and `ANY DEFINED BY` types must be replaced by a reference to an information object class field, and the association table of each `ANY DEFINED BY` type must be replaced by an information object set (see footnote 15 on page 343);

4. try and insert the `AUTOMATIC TAGS` clause in the module header and remove the tags that become useless;

5. try and replace the character string types (`GraphicString` in particular) by `BMPString` or `UniversalString`, even `CHARACTER STRING`; in this case, unfortunately, the encoding is not compatible and such changes induce a new version for the specification;

6. each macro definition should be replaced by an information object class (the `WITH SYNTAX` clause offers a user-defined syntax almost similar to the macro's), a parameterized type or a parameterized value as appropriate (see Section 16.7 on page 374);

7. check if the uses of `EXTERNAL` can be improved with the same encoding when substituting them with the `EMBEDDED PDV` type for example.

These few modifications ensure better readability and upgradeability for the specification; they also make easier the maintenance and automation by ASN.1 compilers. Note that the main part of the updating can be carried out with no change in the encoding, so that interworking is preserved (in this case the two modules can keep the same object identifier as explained on page 163).

The advantages and procedures for the upgrade from ASN.1:1990 to ASN.1:1994/97 have also been detailed in a document coming out of the ITU-T SG VII meeting in Geneva in December 1997. We reproduce it below as directions for use for the specifier:

- *What is the difference between X.208/X.209 and the X.680 & X.690 series of standards?*
  The main difference is that the multitude of defect reports that were issued against X.208 have been corrected. With X.208 there were many ambiguities in the notation that made it possible to write ASN.1 modules which when implemented results in non-interoperability and even though both peers are fully conformant. The most visible manifestations of these bug fixes are the replacement of the macro notation and `ANY`/`ANY`

DEFINED BY types, change in the CHOICE value notation, and the mandatory presence of identifiers in the definition of SET, SEQUENCE and CHOICE types.

- *What is the difference between the X.680 & X.691 series of standards adopted in 1994 versus 1997?*
  The X.680 & X.691 series of standard adopted in December 1997 (referred to below as ASN.1:1997) is a merge of the base version of the 1994 standard and the corrigenda and amendments that were issued between 1994 and 1997. This includes notation to extend types ("...") and the introduction of the character string type, UTF8String.

- *What are some benefits to conforming to ASN.1:1997?*
  Greater readability, precision, flexibility and ease of implementation. For example, when macros are used in X.208 ASN.1 modules, you cannot tell what the macro does without its author having to describe its behavior to you. With ASN.1:1997 the notation is much more precise, making this unnecessary. Indeed, the ASN.1:1997 replacement syntax for ANY DEFINED BY (now called an open type) and macros such as OPERATION is so much better that applications can be written more quickly, simply and in significantly fewer lines of code compared to when X.208 is employed. With the ASN.1:1997 syntax, it is possible for encoders/decoders to fully decode a message, including open types nested in open types nested in open types... (i.e. ANY nested in ANY nested in ANY...), all via a single call to the encoder or decoder, and without any custom modifications to the ASN.1 to accomplish this. Contrast this to the current approach which either requires you to repeatedly call the encoder/decoder for each level of ANY that is to be encoded or decoded, or which requires you to make custom modifications to the ASN.1 and utilize special ASN.1 tools that understand such customizations.

- *Will conformance to ASN.1:1997 increase the complexity of my application?*
  No. However, it can simplify your application and is less likely to result in protocol errors.

- *Will conformance to ASN.1:1997 alter the encoding of messages (the bits on the wire)?*
  No. All applications written to X.208 can be converted to X.680 & X.683 without affecting encodings.

- *Do we have to change already existing applications that use X.208/X.209 if our application standard is changed to use ASN.1:1997?*

  No, there is no need to change existing applications to use ASN.1:1997, for how you choose to implement your application is a local matter. The upgrade is a change in the standards, not the implementation. Consider the fact that today your implementation is considered fully conformant so long as its messages are properly encoded/decoded and it correctly follows all aspects of the protocol it implements, regardless of whether you hand-encode your messages or use an ASN.1 toolkit. Similarly, if one peer chooses to continue using their already deployed application that uses X.208/X.209, and the other peer uses the ASN.1:1997 version of that application's ASN.1 specification, it will be impossible for either peer to know what version of ASN.1 the other is using. This is because any ASN.1 specification converted from X.208 to X.680 & X.683 yields identical encodings.

- *If the application that I wish to implement uses ASN.1:1997, will it be conformant if I change the ASN.1 to X.208 so as to reuse an ASN.1 toolkit that I already have?*
  Yes, it will be conformant. Today very few implementations use ASN.1 from application standards without performing some sort of customization on them so they can be better handled by whatever tool they use. Nonetheless, they have always been considered conformant so long as they fully obey the application protocol. Similarly, if you modify an ASN.1 specification written according to ASN.1:1997 so as to reuse your ASN.1 toolkit (or for whatever reason), your application will be conformant as long as it obeys the application protocol. Note though that the way you currently take care to avoid introducing errors when customizing your ASN.1 is the same that you will need to take care to avoid introducing errors if you choose to alter your ASN.1 specification from ASN.1:1997 to X.208. Since ASN.1:1997 is more descriptive than X.208, little or no customizations are required to use it with conforming ASN.1:1997 toolkits, hence chances of introducing errors due to customizations are reduced when using ASN.1:1997. Contrast this with X.208 specifications for which customizations are often required due to the many ambiguities in the X.208 grammar.

- *What do I need to change in order to conform to ASN.1:1997?*
  See [ISO8824-1] clause A.3, "Migration to the Current ASN.1 Notation", and Annex H, "Superseded Features", for details. In a nutshell, the main changes you will need to make are:

  1. ensure that all the components of `SET`, `SEQUENCE` and `CHOICE` have identifiers.
  2. Include a colon after the identifier in `CHOICE` values.
  3. Change `ANY` and `ANY DEFINED BY` to use the more descriptive open type notation (see Annex H of X.680).
  4. Change the macro notation to the information object class notation (e.g., if `OPERATION` is used) or parameterized types (e.g., if `SIGNED` is used).

- *Is there any help that SG 7 can provide to assist in our better understanding ASN.1:1997 and in modifying our ASN.1 modules to conform to ASN.1:1997?*
  Yes. To assist you in changing your ASN.1 modules to conform to ASN.1:1997 we have arranged the following:

  1. A one day hands-on (no cost) tutorial on the differences between X.208 and ASN.1:1997, details on the benefits of ASN.1:1997, how to convert from X.208 to ASN.1:1997, etc., for editors and whomever else is interested.
  2. The ISO/IEC ASN.1 Rapporteur, John Larmouth, has volunteered to provide whatever assistance is needed to convert your X.208 specifications to ASN.1:1997, including doing the entire editing job if you do not have an editor available to do the work. John's email address is J.Larmouth@salford.ac.uk.
  3. The ASN.1 Editor, Bancroft Scott, has volunteered to check the syntax of your ASN.1 specification to ensure that it conforms to ASN.1:1997, and to answer any questions concerning ASN.1:1997 and converting to it. Bancroft's email address is baos@oss.com.
  4. OSS Nokalva has agreed to make the OSS ASN.1 Tools available at no cost to editors to assist them in verifying that their ASN.1:1997 specifications are valid. Email: baos@oss.com.

With these rules, the standards ACSE [ISO8650-1], Presentation [ISO8823-1] and MHS (ITU-T X.400 series), for example, were rewritten in ASN.1:1994/97 and the version described in the ASN.1:1990 were removed from the ISO and ITU-T catalogs. The ROSE [ISO9072-2] standard and the directory ITU-T X.500 series have been rewritten in ASN.1:1997 but their ASN.1:1990-conformant editions are still available. The updating of the X.700 series of network management standards is now over and should be approved in February 2000.

6.4.3 Migration from ASN.1:1994 to ASN.1:1990

This go-backward case (definitely not recommended!) can unfortunately occur when tools have not been updated for ASN.1:1994. We shall not discuss the matter but refer to [ETSI295].

Chapter 7

Protocols specified in ASN.1

Contents

What a tremendous advantage not to have done anything, but this should be enjoyed with moderation.

Antoine Rivarol.

As a conclusion for this introductory part, we describe a few application domains of the ASN.1 notation. Though wordy it might seem, this chapter is not meant to be a comprehensive description of all the protocols specified with ASN.1 and many other application domains will undoubtedly emerge in the near future.

7.1 High-level layers of the OSI model

As we shall see on page 361, the Presentation Protocol Data Unit (PPDU, 6th layer) is specified with ASN.1 [ISO8823-1]. Some PPDUs (particularly those of connection denial and connection acceptance for Presentation) are described in the module `ISO-8823-PRESENTATION`. Each PPDV is transmitted afterwards as a parameter of a Session primitive (5th layer, see Figure 3.1 on page 18).

The Application layer (7th layer) is divided into service elements, that are standardized for being often used by communicating applications. The data transfer brought about by the service elements are necessarily specified in ASN.1. We can mention:

- the *Association Control Service Element* (ACSE, [ISO8650-1] standard), which manages the establishment and termination of the connections between two distant applications;
- the *Commitment, Concurrency, and Recovery* service element (CCR, [ISO9805-1] standard), which provides a number of cooperation and synchronization task functions in a distributed environment: it makes sure the operation left to a remote application (a database update, for instance) is executed properly, it ensures the information coherence when several processes are running in parallel and re-establishes a clean environment if errors or failures occur;
- the *Remote Operation Service Element* (ROSE, [ISO13712-1] standard): a very general client-server mecanism, which hides from the application programmer the existence of a communication between processes; it can ask the remote application to execute operations or to collect results and errors; each interface's operation is described with ASN.1 as an information object of the `OPERATION` class; ROSE provides a common and standardized method for carrying requests and answers laying by specific gaps in the APDU to be filled in dynamically during communication;
- the *Reliable Transfer Service Element* (RTSE, [ISO9066-2] standard) which can transfer safely and permanently APDUs by taking over the communication where the transfer was interrupted or warning off the sender that the transfer is not possible.

These generic service elements can then be combined more easily to build up applications for which data transfers are also specified in ASN.1 such as:

- the *File Transfer, Access, and Management service* (FTAM, [ISO8571-4] standard) for transferring files or programs between heterogeneous systems. It also provides an access to the files to read or write, to change the rights they have been attributed, or to change their size and content description (equivalent to the Unix `ftp`);
- the *Virtual Terminal service* (VT, ISO 9041 standard) for controlling a terminal that screen, keyboard and some peripheral like a printer for example, without the application knowing all the types of terminal which it may deal with (equivalent to the Unix `telnet`);
- the *Job Transfer and Manipulation service* (JTM, ISO 8832 standard) for executing data processing from a remote machine (a complex computation on a powerful computer for instance), supervising it and getting the results.

7.2 X.400 electronic mail system

E-mail is probably one of today's most famous information technology applications. It is therefore worthwhile describing what is meant by this expression. As exposed in our history review on page 60, it had a most important role for ASN.1 because the X.208 standard (ASN.1 first edition) directly resulted from the X.409:1984 notation, which had been designed for representing the various parts of an e-mail. Indeed, as the eighties saw the use of e-mail become more common, the CCITT was led to standardize an OSI-compliant e-mail service.

Today, the most industrialized countries have an e-mail public service that conforms to the X.400 standard service[1]. Such a service promotes the development of communicating applications particularly in office

[1]The equivalent ISO standard is called MOTIS (*Message Oriented Text Interchange System*) recorded as ISO 10021. Information about the X.400 standard services can be found at http://ftp.net-tel.co.uk/iso-iec-jtc1-sc33-wg1/ and http://www.alvestrand.no/x400/.

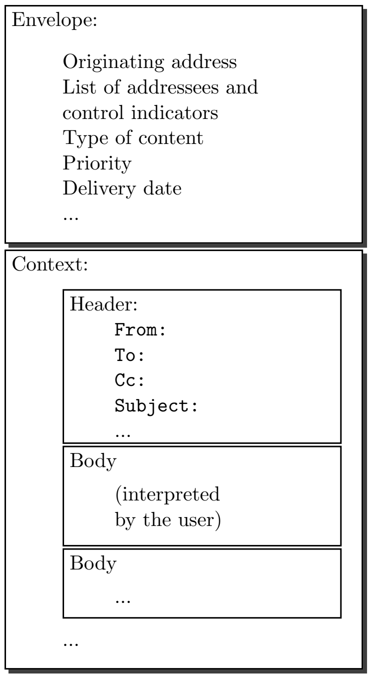


Figure 7.1: Structure of an X.400 message

automation and computerized documents (see Section 7.6 on page 88). The X.400 standards define the message format, given in Figure 7.1, and the exchange protocol but the message content is up to the user, and therefore outside the boundaries of the OSI world.

This standard series, which consists of about 5,000 lines of ASN.1 notations was completely rewritten to be compliant with all the functionalities of the ASN.1:1994 edition. The extract in Figure 7.2 on the next page is the ASN.1 definition (slightly simplified) of the envelope of message delivery according to Figure 7.1. The description of ASN.1 concepts in the previous chapter should be sufficient to make out the data types of this envelope.

Compared to the *Simple Mail Transfer Protocol* (SMTP), the e-mail protocol on the Internet, a BER-encoded X.400 message is more compact and offers more security since it is possible to ask for an acknowledgement of receipt and reading. On the other hand, SMTP fans may argue that BER encoding, which requires a decoder, is not as readable as the SMTP ASCII encoding, which transfer characters 'as is'.

```
MessageSubmissionArgument ::= SEQUENCE {
  envelope  MessageSubmissionEnvelope,
  content   Content }
MessageSubmissionEnvelope ::= SET {
  originator-name        OriginatorName,
  original-encoded-information-types
    OriginalEncodedInformationTypes OPTIONAL,
  content-type           ContentType,
  content-identifier     ContentIdentifier OPTIONAL,
  priority               Priority DEFAULT normal,
  per-message-indicators PerMessageIndicators
                               DEFAULT {},
  deferred-delivery-time [0] DeferredDeliveryTime
                               OPTIONAL,
  per-recipient-fields   [1] SEQUENCE
    SIZE (1..ub-recipients) OF
    PerRecipientMessageSubmissionFields }
```

Figure 7.2: An extract of ASN.1 for an X.400 message

7.3 X.500 Directory

The directory (ITU-T X.500 recommendation series or ISO 9594 standards)[2] is an international and distributed database that can store any kind of information about persons, organizations, communicating application entities, terminals, mailing lists, etc. It is often described in parallel with the X.400 e-mail because it provides an interactive search of subscriber addresses but also other items of information like phone number, address, favorite medium (e-mail, fax, phone...), photography, public key encoding...

The X.500 directory is a hierarchical database. Every node of this international tree is identified with a number of standardized or locally defined attributes; it can be referenced by a unique distinguished name, which locates it within the tree. Powerful search requests using pattern matching with the attributes' values enable to implement the directory with a user-friendly interface.

ASN.1 is fully used for the X.500 directory, particularly to specify the requests and the modification of the *Directory Access Protocol* (DAP) attributes. Figure 7.3 on page 85 defines the information object class `ATTRIBUTE` which allows a description of each attribute (data types,

[2] http://www.dante.net/np/ds/osi.html, http://www.cenorm.be/isss/Workshop/DIR/Default.htm

applicable comparison rules, usage) and the class `MATCHING-RULE`, which is used to define compatibility rules between attributes (for example, case-insensitive comparisons to differentiate names).

Of course, the class `MATCHING-RULE` only defines the comparison function interface, for their implementation is down to each provider of the directory service.

For a more thorough description of the directory, the reader can refer to [Cha96]. The information object classes `ATTRIBUTE` and `MATCHING-RULE` above mentioned are used in Chapter 15. Finally it is worthwhile mentionning that the ASN.1 specifications of the X.500 service protocols are being adapted for the Internet.

7.4 Multimedia environments

A growth industry because of the Web or digital phone networks, the multimedia applications also benefit from standards formalized in ASN.1. MHEG (*Multimedia and Hypermedia information coding Expert Group*[3], ISO 13522 standard) uses an object-oriented approach to describe the representation of multimedia and hypermedia information for exchanging it between applications (using the Distinguished Encoding Rules, DER).

Numerous application domains for the MHEG standard are being considered such as interactive digital TV programs, pay-per-view, simulation games, tele-teaching, tele-shopping and many other services where real-time transfer and a regular updating of many multimedia objects are necessary.

There are eight MHEG object classes that are defined both in ASN.1 and in SGML (*Standard Generalized Markup Language*). These classes can transparently exchange objects encoded in many different formats (JPEG, MPEG, text...), including all proprietary formats. The MHEG objects can be icons or buttons to trigger actions when clicked. They are independent from the applications as well as from presentation supports.

In the domain of videoconferencing, which annual growth is huge particularly because the productivity gains induced make it grow in popularity among businessmen, the ITU-T T.120[4] recommendation series describes a multithread architecture of data communication within

[3]http://www.fokus.gmd.de/ovma/mug/

[4]http://www.databeam.com/ccts/t120primer.html

```
ATTRIBUTE ::= CLASS {
  &derivation            ATTRIBUTE OPTIONAL,
  &Type                  OPTIONAL,
  &equality-match        MATCHING-RULE OPTIONAL,
  &ordering-match        MATCHING-RULE OPTIONAL,
  &substrings-match      MATCHING-RULE OPTIONAL,
  &single-valued         BOOLEAN DEFAULT FALSE,
  &collective            BOOLEAN DEFAULT FALSE,
  &no-user-modification  BOOLEAN DEFAULT FALSE,
  &usage                 Attribute-Usage
                           DEFAULT userApplications,
  &id                    OBJECT IDENTIFIER UNIQUE }
WITH SYNTAX {
  [SUBTYPE OF                &derivation]
  [WITH SYNTAX               &Type]
  [EQUALITY MATCHING RULE    &equality-match]
  [ORDERING MATCHING RULE    &ordering-match]
  [SUBSTRINGS MATCHING RULE  &substrings-match]
  [SINGLE VALUE              &single-valued]
  [COLLECTIVE                &collective]
  [NO USER MODIFICATION      &no-user-modification]
  [USAGE                     &usage]
   ID                        &id }

Attribute ::= SEQUENCE {
  type   ATTRIBUTE.&id ({SupportedAttributes}),
  values SET SIZE (1..MAX) OF
    Attribute.&Type ({SupportedAttributes}{@type})}

MATCHING-RULE ::= CLASS {
  &AssertionType OPTIONAL,
  &id            OBJECT IDENTIFIER UNIQUE }
WITH SYNTAX {
  [SYNTAX        &AssertionType]
   ID            &id }

caseIgnoreSubstringsMatch MATCHING-RULE ::= {
  SYNTAX   SubstringAssertion
  ID       id-mr-caseIgnoreSubstringsMatch }

SubstringAssertion ::= SEQUENCE OF CHOICE {
  initial [0] DirectoryString {ub-match},
  any     [1] DirectoryString {ub-match},
  final   [2] DirectoryString {ub-match} }
ub-match INTEGER ::= 128
```

Figure 7.3: Two information object classes defined in the X.500 directory

the environment of a multimedia conference. It describes the establishment of phone meetings regardless of the underlying networks and the exchange of any format of information (binary files, fixed images, notes...) during the meeting. The data protocol is obviously specified in ASN.1 and the encoding compliant with the Packed Encoding Rules (PER).

Many other protocols in multimedia are specified with ASN.1. For example, audiovisual and multimedia systems (ITU-T H.200 series), videophone over RNIS (ITU-T H.320 recommendation), real-time multimedia communication over the Internet (ITU-T H.225, H.245, H.323 recommendation)[5] and fax over the Internet (ITU-T T.38 recommendation)[6] have been regularly mentioned in the press lately.

7.5 The Internet

In the booming Internet world (it is estimated that 25% of the telephone traffic have moved on the Internet by 2003), ASN.1 has appeared for quite a long time now in many *Requests For Comments*[7] (RFC) that specify the Net protocols. RFC 1189[8] (*The Common Information Services and Protocols for the Internet, CMOT and CMIP*) and RFC 1157[9] (*A Simple Network Management Protocol*, SNMP) for example are two alternative protocols allowing a network to control and evaluate the performance of a remote network element.

Unfortunately, in retaining its space of freedom, the *Internet Engineering Task Force* (IETF) has often let itself be entangled in liberal usages of ASN.1. The main critics about RFC are the following:

- the systematical use of `OCTET STRING` to modelize ill-known data or to avoid specifying too formally what they stand for;

- the definition of many macros and macro instances to represent semantic links instead of information object classes and information objects although no ASN.1 compiler properly takes into account

[5] http://www.openh323.org/standards.html, http://people.itu.int/~jonesp/iptel/

[6] http://www.dialogic.com/company/whitepap/4631web.htm

[7] It is a fast way of proposing new standards for the Internet and receive comments (http://www.rfc-editor.org/overview.html).

[8] http://www.faqs.org/rfcs/rfc1189.html

[9] http://www.faqs.org/rfcs/rfc1157.html, http://www.simple-times.org, http://www.snmp-products.com/REF/ref.html

the macro concept (on the other hand, no compiler of the public domain does with the information object class concept unfortunately);

- it does not clearly define the ASN.1 version it uses and mixes up ASN.1:1990 and ASN.1:1994 features, which can result in tricky compilations;
- the liberties they take with ASN.1 syntax and sometimes with the BER encodign rules: this disrecpect any compilation by commercial tools or any use of the generic encoder and decoder they produce. This often leads to hand-made specification implementation.

Two important projects have been recently specified with ASN.1 even though the Internet community is generally quite reserved about such specifications (mainly because they are tagged with the ISO and OSI labels!).

Since its creation in 1992, the ANSI Z39.50 protocol (ISO 10163-1 standard)[10] is specified in ASN.1 and encoded with BER. A variant protocol was used in the WAIS service (*Wide Area Information Server*) to make all kinds of information accessible on the Internet (library catalogs, directories, `ftp` archives, newsgroups, images, source codes, multimedia documents). It provides facilities for keyword search, for extending a search by including new criteria to be applied to the documents already found and for downloading selected documents. The Z39.50 protocol is mainly used in libraries and information centers because it is well-suited for the note formats they deal with. New encoding rules called XER (see Section 21.5 on page 458) are still under construction to promote the use of this protocol on the Web.

The authentication and distribution systems *Kerberos*[11] developed by the *Massachusetts Institute of Technology* (MIT) is a software designed for securing data exchanges within the network of a university or an organization. Since its fifth version, the data transfers are specified in ASN.1. Microsoft has already announced that this authentication system would be supported by *Windows NT 5*®.

Similarly the *Public Key Cryptography Standard* PKCS[12] no. 7 [RSA93] describes with ASN.1 the syntax of encrypted messages with

[10] http://mda00.jrc.it/z39.50/z39.50-asn1.html

[11] http://www.mit.edu/afs/athena/astaff/project/kerberos/www/

[12] http://www.rsa.com/rsalabs/pubs/PKCS/, http://www.rsa.com/rsa/developers/

digital signature encoded in BER. The standard was produced in 1991 jointly by a consortium of computer manufacturers and the MIT.

Finally, business and electronic transaction protocols described in Section 7.7 on the next page, as well as multimedia communications presented in the previous section, are now increasing on an unprecedented scale with the Internet.

7.6 Electronic Data Interchange Protocols (EDI)

The automation of exchange in the legal, economic and business domains now removes needless manual data capture. In order to take advantage of such practice, several standards offer information structures for the documents exchanged.

The *Office Document Architecture* or ODA proposes to transmit in the content of the e-mail the format tags in addition to the text itself so that the addressee could browse the document according to the representation required by the sender. It is particularly suited for office procedures such as word processing, archives and document exchanges.

Specified in ASN.1, the exchanged format called ODIF (*Office Document Interchange Format*, ISO 8613-5 standard or ITU-T T.415 recommendation), enables the transfer of the document description (letter, report, invoice...) and their content (text, graphs, images...) via an X.400 e-mail.

Recommendation ITU-T X.435 proposes an EDI e-mail system above the X.400 e-mail. It is aimed at users of the EDIFACT standard for business document exchange (see Section 24.5 on page 492) and other common EDI syntaxes.

The *Document Transfer And Manipulation* standard (DTAM, ITU-T T.431 recommendations) provides a service for processing, access, management and transfer of documents structured according to the ODA architecture when associating two applications. This service is general enough to cover a wide diversity of telematic application demands such as the group IV fax (ISDN transmission, 5 seconds per page, colour option) and videotext sytems. These two standards are specified in ASN.1.

7.7 Business and electronic transactions

Another one of today's booming area thanks to the generalization of the Internet at home or at work, is that of electronic business[13]. In this context, the transaction security must free up from the diversity of payment media, networks and softwares, and ASN.1 accedes to these requirements.

SET (*Secured Electronic Transaction*)[14] is a standard made up jointly by several american companies (*Mastercard*, *Visa*, *American Express*, *Netscape*, IBM...) in order to secure financial exchange on the Internet. It is based on the PKCS no. 7 standard of public encryption described on page 87 and on the procedure [X.509] for the directory seen in Section 7.3 on page 83. It provides the following services: confidentiality of the information to the transaction, integrity of the transferred data, authentication of the account owner and of the business party.

In order to benefit from the French specificity (France was the first country where the use of chip-based cards, opposed to magnetic track-based cards, were generalized) a national organization called *GIE Cartes Bancaires*[15], in charge of defining card specificities for that country, developed a promising standard, adapted from the SET standard and called C-SET[16] (*Chip-SET*). Also specified in ASN.1, it relies on the card itself to secure the transaction and therefore avoids exchanging authentication certificate.

In the USA, the ANSI X.9[17] committee, which numbers more than 300 members (banks, investors, software companies, associations) is responsible for developing national standards to facilitate financial operations: electronic payment on the Internet, secured service for on-line banks, business messages, fund transfers, etc. All the standards describing these data transfers are specified in ASN.1.

7.8 Use in the context of other formal notations

ASN.1 is the data typing language in three standard formal notations that will be described more thoroughly in Chapter 23.

[13]http://www.ecom.cmu.edu/resources/elibrary/epaylinks.shtml
[14]http://www.setco.org/set_specifications.html
[15]http://www.cartes-bancaires.com, http://www.visa.com/nt/ecomm/set/main.html
[16]http://www.europayfrance.fr/fr/commerce/secur.htm
[17]http://www.x9.org

The *Guidelines for the Definition of Managed Objects* (GDMO, [ISO10165-4] standard) are used to model system administration and technical management aspects as managed objects made of attributes (whose types are described in ASN.1) and actions that modify the attributes' values (the operation arguments and return values are also typed in ASN.1).

System or network management ensures a reliable and continuous functioning while optimizing the performance and make up for hardware failures. Every part of a computer or telecommunication network can be monitored: routers, queues, sensors, logs, software versions, clocks, accounts...

The *Common Management Information Protocol* (CMIP, [ISO9596-1] standard) in charge of the bi-directionnal dispatching of all the management information (the managed objects) between the manager and the agents is specified in ASN.1.

SDL (*Specification and Description Language*, ITU-T Z.100 recommendation) formalizes various concepts of telecommunication networks: signalling, switching network inter-operability, data processing, protocols... SDL is a very popular language and its scope of action goes beyond the telecommunication area.

It was at first related to the language *ACT ONE* for describing the data types handled by the specification but afterwards turned to ASN.1 to take over the task thereby making the development of a protocol easier: formalization, implementation, validation and tests. ASN.1 now tends to gradually take over *ACT ONE* more and more often.

TTCN (*Tree and Tabular Combined Notation*, ISO 9646 standard) is a test description language particularly convenient for protocol tests. It can describe a collection of abstract tests (regardless of the architecture) as PDUs or service primitives (see footnote 1 on page 22) without paying attention to the encoding. ASN.1 was included in TTCN in order to make it possible to describe test sequences for the application layer protocols; these sequences can be used even if the specification to be tested is not written in ASN.1.

7.9 Yet other application domains

We conclude this chapter with an enumeration *à la Prévert*[18] of other uses for ASN.1.

The forthcoming *Aeronautical Telecommunication Network* (ATN), which should be operational in Europe around 2005, will be based on OSI protocols. The information exchanged between planes and ground control systems will be specified in ASN.1 and encoded in PER.

In the telecommunication domain, ASN.1 is essential although all the branches have not adopted it yet. It is indeed used for mobile phones (with the *Mobile Application Part* or MAP protocol for the GSM networks, or the third generation mobiles conform to the UMTS[19] standard), the free phone numbers, the *Integrated Services Digital Network* (ISDN), the intelligent networks (the *Intelligent Network Application Protocol*, also called INAP, whose second capability set (CS2) contains more than 250 pages of ASN.1 assignments and uses the X.500 directory definition), the *Signalling System No. 7* (SS7) between switches (signalling is an area where the use of ASN.1 should be generalized)...

The duality telephony/information technology, which allows communication between a phone system (such as a *Private Automatic Branch eXchange*, PABX) with computing applications, is undergoing radical changes, particularly in phone exchange centers whose productivity is tremendously improved or in office automation where it can integrate in a more homogeneous way all the computing services and products. The *Computer Supported Telecommunications Applications* (CSTA[20]) standards specify the structure of the message exchanged between equipments and computing applications in ASN.1 using a BER encoding.

The MMS (*Manufacturing Message Specification*, ISO 9506 standard) allows to control manufacturing without having to care about the potential heterogeneity of equipments: robots, digitally controlled machine tool, bespoke programmed automaton. It is used in exchanges for selling or buying electricity in real time, for controls of paper mills, for car assembly lines and for chemical or food factories.

[18]Other enumerations can be found at http://www.oss.com/rstand.htm, http://www.inria.fr/rodeo/personnel/hoschka/baosmsg.txt and http://www.inria.fr/rodeo/personnel/hoschka/ralphmsg.txt.

[19]http://www.umts-forum.org/, http://www.3gpp.org/

[20]http://www.etsi.org/brochures/stateart/huff.htm

The market of telematics applied to transport information and control systems[21] will be booming for the next twenty years. The progress of navigation systems by satellite, of digital cartography and mobile telecommunications may enable optimizing the management of taxi or public transport vehicle fleets and smooth the road traffic with intelligent signals and information transmission to individual navigation systems. Some protocols of the intelligent transport domain are specified in ASN.1 and encoded in PER (see Section 21.3 on page 456).

The Radio-Frequency IDentification (or RFID[22]) is implemented in numerous industrial sectors (person or vehicle identification, stock management...). The electronic tags are made of miniaturized radio transmitters that can be accessed from a few centimeters to several meters far or through obstacles (thereby forbidding barcodes for instance). The PER encoding seems to be an excellent answer to bandwidth problems frequently encountered in this area.

In the USA, the *National Center for Biotechnology Information* (NCBI) created *GenBank*[23], a database featuring around four million DNA sequences (DesoxyriboNucleic Acid). Everyday the american center gives and receives DNA sequences from its European and Japanese counterparts[24]. The *National Library of Medicine* also designed four databases (*Unified Medical Language System*, UMLS[25]) whose exchange format are specified in ASN.1. They describe scientific papers among other things.

The ISO/CEI 7816-4 standard[26] use a BER encoding for exchanging data with integrated circuit(s) cards with contacts; one of today's application domain is the Social Security electronic card. The european project Netlink on interworking social security card systems relies on ASN.1 for describing the card's data structures. The technical committee TC 251[27] in charge of Health Informatics at the European Committee for Standardization (CEN) published the ENV 12018 standard on "Identification, administrative, and common clinical data structure

[21] Some elements of information can be found at http://www.iso.ch/meme/TC204.html or http://www.nawgits.com/icdn.html.

[22] http://www.aimglobal.org/technologies/rfid/resources/papers/rfid_basics_primer.html

[23] http://www.ncbi.nlm.nih.gov/Web/Genbank/index.html

[24] ftp://ncbi.nlm.nih.gov/mmdb/specdocs/ncbi.asn

[25] http://www.nlm.nih.gov/research/umls/

[26] http://www.iso.ch/cate/d14738.html

[27] http://www.centc251.org/

for Intermittently Connected Devices used in healthcare" for which the data structures are described in ASN.1.

It is now up to the reader[28] to add on to the list other original applications of ASN.1 and even correct some inaccuracies in those already in this chapter.

[28]By sending an e-mail to asn1@rd.francetelecom.fr.

Part II

User's Guide and Reference Manual

Chapter 8

Introduction to the Reference Manual

Contents

Make everything as simple as possible, but not simpler.

Albert Einstein.

The User's Guide and Reference Manual constitute the main part of this book. Using as many examples as possible we shall present all the features of ASN.1. The chapters are ordered by increasing difficulty. Some chapters can be discarded depending on your proficiency and experience with ASN.1: see the reading directions of Figure 1 on page xxi.

8.1 Main principles

We have wanted this second part of the book to be both a User's Guide and a Reference Manual in order to comply with the largest audience's need. For doing so each section is systematically divided into two subsections. The first, always called 'User's Guide', is more particularly aimed at the beginner since this is where we introduce the concepts of ASN.1 and gradually increase the difficulty of our examples to span all the possibilities offered by the notation. We often conclude with some examples drawn from real protocols specifications. Once understood what we explain in this first sub-section, the reader may start with the second but we recommend to leave it aside at first.

The second sub-section is always called 'Reference Manual' and is aimed at the proficient specifier or the programmer. It is not meant to be read completely but to be referred to for looking up a particular point of ASN.1 syntax or semantics. Indeed, this part relies on the ASN.1 grammar (presented in extended Backus-Naur notation) to detail, with generally short rules, what is allowed or forbidden in the standard. These few editorial conventions help to write semantic rules that are simpler and easier to understand.

Though many rules have been reproduced in several sections to obtain the information that is looked faster, train of these detailed descriptions cannot be sequential or strictly top-down because grammar productions and semantic rules frequently call one another (if necessary, refer to the index on page 543 to find a specific production by its name). The semantic rules obviously recall the clauses of the ASN.1 standards but we have included in our descriptions many enlightments due to our own experience, to discussions on newsgroups or electronic correspondence with Bancroft Scott and John Larmouth, respectively editor and rapporteur of the ASN.1 standard at ISO and ITU-T.

The elements of the 'User's Guide' do not obviously encompass all the rules of the 'Reference Manual'. And vice versa: the 'User's Guide' includes tips for writing specifications that do not necessarily appear in the 'Reference Manual'.

8.2 Editorial conventions

First, we define a few terms of parsing theory [ASU86]. A grammar includes the following components:

- a set of *lexical tokens* or *terminals* (which are sent by the scanner (or lexical analyzer) to the parser, see Figure 22.1 on page 464);

- a set of productions, where each production or re-writing rule is made of a non-terminal (called 'left-hand side of the production'), an arrow "→", and a series of lexical tokens and non-terminals (called 'right-hand side of the production'). For clarity's sake, productions with the same left-hand side are grouped and separated by the "|" symbol, which can be read 'or'. We call *production* (singular), a collection of re-writing rules with the same left-hand side and *alternative*, each one of these re-writing rules. We keep the term '*rule*' for the semantic rules in natural language that are detailed after each production.

The following editorial conventions are systematically used in the second part of the book.

- the lexical tokens of ASN.1 are written in lower-case letters in a sans serif font (e.g. typereference);
- the keywords (except the character string types) are in upper-case letters and in `teletype font` (e.g. `BOOLEAN`); special characters and symbols are between double quotes in `teletype font` (e.g. "`,`");
- the non-terminals begin with an upper-case letter and are denoted *in italics* (e.g. *Assignment*);
- a non-terminal whose production appear in another section is underlined (e.g. *DefinedValue*); this definition can be easily found using the index on page 543;
- we use the Greek letter *epsilon* 'ϵ' to state self-explicitly an empty alternative in a production, i.e. an alternative for which no information should be given;
- the recursive items are denoted by the rational operators "*" and "+" which are defined with the symbol "$\cdots$" as follows (where c stands for any separation mark such as "`,`" or "`.`"):

$$\begin{array}{lcrl|lcrl} X \rightarrow A\ \mathsf{c} \cdots^{*} & \Longleftrightarrow & X \rightarrow & A\ Y & X \rightarrow A\ \mathsf{c} \cdots^{+} & \Longleftrightarrow & X \rightarrow & A\ Y \\ & & | & \epsilon & & & Y \rightarrow & \mathsf{c}\ A\ Y \\ & & Y \rightarrow & \mathsf{c}\ A\ Y & & & | & \epsilon \\ & & | & \epsilon & & & & \end{array}$$

- in order to facilitate the reading, some productions appear in several sections (for example, the production *NamedValue* appears in the descriptions of *SequenceValue* and *SetValue*);

- the semantic rules are all referenced by a number in angles "⟨ ⟩" to spot them easily;

- the yellow parts indicate work under discussion at the ISO|ITU-T ASN.1 working group and not standardized yet, as it is. These parts should be *handled with care*: it is indeed impossible to ensure that these propositions will be adopted in the end. However, they illustrate the fact that the ASN.1 standard is not frozen and try to take into account all users' requests.

8.3 Lexical tokens in ASN.1

8.3.1 User's Guide

In this section, we present the lexical tokens of the ASN.1 notation, which are called 'ASN.1 items' in the standard.

From the lexical analyzer's viewpoint [ASU86], two *lexemes* (the value taken by a lexical token) are separated by a space, a comment or a newline except if the first character of the second lexeme is not allowed for the lexical token it refers to[1].

8.3.2 Reference Manual

bstring

⟨1⟩ A binary string consists of a (possibly empty) series of `0`, `1`, spaces, tabulations and newlines, preceded by the character "`'`" and followed by "`'B`".

⟨2⟩ The spaces, tabulations and newlines (of code 9, 10, 13 and 32 in Table 11.2 on page 178) appearing in a binary string are meaningless. These characters were not allowed in ASN.1:1990.

[1]For ASN.1:1994 and ASN.1:1997, the valid characters (except for comments where all characters are allowed and character strings whose character set depends on their governing type) are: "`A`" to "`Z`", "`a`" to "`z`", "`0`" to "`9`", "`:`", "`=`", "`,`", "`{`", "`}`", "`<`", "`.`", "`@`", "`(`", "`)`", "`[`", "`]`", "`-`", "`'`", "`"`", "`|`", "`&`", "`^`", "`*`", "`;`" and "`!`".

During the January 1999 meeting in Lannion, it has been proposed to include the underscore "_" as an equivalent to the dash "-" in references and identifiers (particularly to make ASN.1 consistent with other formal notations it is used with, such as SDL and TTCN, see Chapter 23). During the June 1999 meeting in Geneva, the working group preferred to postpone the decision because they aspired to a solution that would not jeopardize existing specifications and keep their homogeneity.

comment

⟨3⟩ A comment begins with a double dash "`--`" and is made of an unrestricted number of (any) characters and stops at the following newline or the next double dash.

⟨4⟩ When applied to a constructed type like `SEQUENCE`, `SET`, `CHOICE`, `SEQUENCE OF` or `SET OF`, a comment may use the meta-notation "`@`" presented on page 231.

⟨5⟩ Comments in natural language occurring in a specification are ignored by the ASN.1 tools: they are used only to explain the meaning to be associated with a type by the communicating application. However some compilers do take into account formal comments called *directives* (or sometimes *encoding pragmas*), which generally begin with special characters depending on the tool such as "`--*`", "`--$`" or "`--<    >--`" (see Chapter 22).

cstring

⟨6⟩ A character string consists of a (possibly empty) series of characters of the repertoire denoted by the associated character string type (see Chapter 11). It begins and stops with a double quote "`"`".

⟨7⟩ If a string includes a double quote, this double quote should be preceded by another double quote (see the example on page 173).

⟨8⟩ If the string is spread out over several lines, the spaces and tabulations that precede and follow the newline character are meaningless (the newlines themselves are ignored). In ASN.1:1990, strings on more than one line were not allowed.

⟨9⟩ Strings cannot include control characters, except for the type `IA5String` if the production *Tuple* is used (see on page 197) and for the types `UniversalString`, `BMPString` and `UTF8String` if the production *Quadruple* is used (see on page 196).

hstring

⟨10⟩ A hexadecimal string consists of a (possibly empty) series of characters "`A`" to "`F`" (upper-case), "`0`" to "`9`", space and newline, beginning with "`'`" and followed by "`'H`".

⟨11⟩ The spaces, tabulations and newlines (of code 9, 10, 13 and 32) appearing in a hexadecimal string are meaningless. These characters were not allowed in ASN.1:1990.

identifier

⟨12⟩ An identifier consists of a lower-case letter followed (or not) by several letters (upper-case letters from "`A`" to "`Z`" and lower-case letters), digits and dashes (the last character cannot be a dash and the double dash is not allowed).

⟨13⟩ The underscore "_" is not allowed in identifiers (see footnote 1 on page 100).
⟨14⟩ The length of an identifier is not limited but too long identifiers may be rejected by some target computing languages used by ASN.1 compilers.
⟨15⟩ The identifiers are used to label the components of `SEQUENCE` and `SET` types, the alternatives of `CHOICE` types, the numbers of `INTEGER` and `ENUMERATED` types, and the bit locations in `BIT STRING` types.
⟨16⟩ In case a national standardization organization defines a standard derived from the ASN.1 standard to use a case-unsensitive language (for example, SDL, see Section 23.1 on page 476), another way of distinguishing the lexical tokens identifier and valuereference from typereference and modulereference should be agreed on (for example, adding a semicolon at the end of each definition as suggested in rule ⟨37⟩ on page 104).

modulereference

⟨17⟩ The lexical token modulereference consists of the same character sequence as typereference (see rules ⟨26⟩ and ⟨28⟩ on the next page).

number

⟨18⟩ A number consists of one or several digits. If the number has more than one digit, the first should be different from `0`.

objectclassreference

⟨19⟩ The lexical token objectclassreference consists of the same character sequence as typereference (see rules ⟨26⟩ and ⟨28⟩ on the next page), except that lower-case characters are not allowed.

objectfieldreference

⟨20⟩ The lexical token objectfieldreference consists of the ampersand "`&`" followed by an objectreference (see rule ⟨21⟩ on the current page).

objectreference

⟨21⟩ The lexical token objectreference consists of the same character sequence as valuereference (see rule ⟨32⟩ on the next page).

objectsetfieldreference

⟨22⟩ The lexical token objectsetfieldreference consists of the ampersand "`&`" followed by an objectsetreference (see rule ⟨23⟩ below).

objectsetreference

⟨23⟩ The lexical token objectsetreference consists of the same character sequence as typereference (see rules ⟨26⟩ and ⟨28⟩ on the next page).

SignedNumber → number
| "-" number

⟨24⟩ The value "-0" is not allowed.

typefieldreference

⟨25⟩ The lexical token typefieldreference consists of the ampersand "&" followed by a typereference (see rules ⟨26⟩ and ⟨28⟩ below).

typereference

⟨26⟩ The lexical token typereference consists of an upper-case letter, followed (or not) by several letters (upper-case letters from "A" to "Z", and lower-case letters), digits and dashes (the last character cannot be a dash and the double dash is not allowed).

⟨27⟩ The length of a typereference is unlimited but too long references may not be supported by some target computing languages used by ASN.1 compilers.

⟨28⟩ typereference cannot be one of the following keywords: `ABSENT`, `ABSTRACT-SYNTAX`, `ALL`, `APPLICATION`, `AUTOMATIC`, `BEGIN`, `BIT`, `BMPString`, `BOOLEAN`, `BY`, `CHARACTER`, `CHOICE`, `CLASS`, `COMPONENT`, `COMPONENTS`, `CONSTRAINED`, `CONTAINING`, `DEFAULT`, `DEFINITIONS`, `EMBEDDED`, `ENCODED`, `END`, `ENUMERATED`, `EXCEPT`, `EXPLICIT`, `EXPORTS`, `EXTENSIBILITY`, `EXTERNAL`, `FALSE`, `FROM`, `GeneralizedTime`, `GeneralString`, `GraphicString`, `IA5String`, `IDENTIFIER`, `IMPLICIT`, `IMPLIED`, `IMPORTS`, `INCLUDES`, `INSTANCE`, `INTEGER`, `INTERSECTION`, `ISO646String`, `MAX`, `MIN`, `MINUS-INFINITY`, `NULL`, `NumericString`, `OBJECT`, `ObjectDescriptor`, `OCTET`, `OF`, `OPTIONAL`, `PATTERN`, `PDV`, `PLUS-INFINITY`, `PRESENT`, `PrintableString`, `PRIVATE`, `REAL`, `RELATIVE-OID`, `SEQUENCE`, `SET`, `SIZE`, `STRING`, `SYNTAX`, `T61String`, `TAGS`, `TeletexString`, `TRUE`, `TYPE-IDENTIFIER`, `UNION`, `UNIQUE`, `UNIVERSAL`, `UniversalString`, `UTCTime`, `UTF8String`, `VideotexString`, `VisibleString`, `WITH`.

⟨29⟩ The underscore "_" is not allowed in typereference (see footnote 1 on page 100).

⟨30⟩ Rule ⟨16⟩ on the preceding page applies also to typereference.

valuefieldreference

⟨31⟩ The lexical token valuefieldreference consists of the ampersand "&" followed by a valuereference (see rule ⟨32⟩ below).

valuereference

⟨32⟩ The lexical token valuereference consists of the same character sequence as identifier (see rule ⟨12⟩ on page 101).

valuesetfieldreference

⟨33⟩ The lexical token valuesetfieldreference consists of the ampersand "`&`" followed by a typereference (see rules ⟨26⟩ and ⟨28⟩ on the page before).

word

⟨34⟩ The lexical token word consists of an upper-case letter, followed (or not) by several upper-case letters, digits and dashes (the last character cannot be a dash and the double dash is not allowed).

⟨35⟩ word cannot be one of the keywords quoted in rule ⟨9⟩ on page 326.

"`;`"

⟨36⟩ The semicolon "`;`" is never used in an ASN.1 specification except at the end of the `EXPORTS` and `IMPORTS` clauses (see on page 115).

⟨37⟩ The semicolon "`;`" is used at the end of all ASN.1 assignments defined in the context of the SDL formal language; this is necessary to remove all syntactic ambiguities since SDL does not make case differences (see rule ⟨16⟩ on page 102).

Chapter 9

Modules and assignments

Contents

[...] I call 'even number' a number that can be equally divided by two. *This* is a geometrical definition.

Blaise Pascal, *On Geometrical Spirit and the Methods of Geometrical Demonstrations, i.e. Methodical and Flawless.*

The other chapters of this Reference Manual will frequently refer to the current chapter, most of the time implicitly. Indeed we present here the various categories of assignments that can be specified in ASN.1 and we expose how these should be collected in modules or how they can be referenced.

This chapter is very unlikely to be the most entertaining of this text; therefore we would suggest the reader to go back over the case study of Chapter 4 before proceeding further with this guide because the guidelines to be found there will not be repeated hereafter.

9.1 Assignments

9.1.1 User's Guide

Some concepts of ASN.1 were only breached in the case study of Chapter 4 or in Chapter 5. There are six distinct categories of assignments altogether that can be found in a module body: types, values, value sets, information object classes, information objects and information object sets.

To define a new type we should give it a name beginning with an upper-case letter (conform to the lexical token typereference defined on page 103), followed by the symbol "::=", and a type expression conform to what is exposed in Chapters 10 and 11 for the basic types, in Chapter 12 for the constructed types and in Chapter 13 for the subtype constraints:

```
TypeReference ::= CHOICE { integer INTEGER,
                           boolean BOOLEAN }
```

Other examples of type definitions were given in Section 5.2 on page 45.

Type is the most important concept in ASN.1 since it enables to specify values that can be transferred between two communicating applications (indeed a type can be semantically interpreted as a set of values). As a result, any ASN.1 assignment that would not be referenced (even indirectly) in a type assignment would not be involved in the communication process.

The type of highest level, which is not referenced in another type assignment, is called PDU (Protocol Data Unit). This is the type of the data that are exchanged between applications.

To define a (abstract) value we should give it a name starting with a lower-case letter, followed by its type generally referenced by its name (starting with an upper-case letter), the "::=" symbol and the value:

```
value-reference TypeReference ::= integer:12
```

Other value definitions were presented in Section 5.3 on page 48.

An abstract value is never encoded (i.e. it will never be transmitted to the remote application), it is 'only' used to refine the abstract syntax. It can be found in the `DEFAULT` clauses of a `SEQUENCE` or `SET` type (see Section 12.2 on page 218 and Section 12.3 on page 226), in subtype constraints (see Chapter 13) and in information object classes or information object definitions (see Chapter 15). A value definition should therefore always be referenced by another definition, but the specifier sometimes uses it formulated as a sort of formal comment to give an example of value for some types of the specification.

If two types (whether basic or constructed) are syntactically identical, every value of one type can be governed by the other type. In other words, if two types are *compatible*, every value of one type can be used instead of a value of the other:

```
Pair ::= SEQUENCE { x INTEGER,
                    y INTEGER }
Couple ::= SEQUENCE { x INTEGER,
                      y INTEGER }
pair Pair ::= { x 5, y 13 }
couple Couple ::= pair
Lighter-state ::= ENUMERATED { on(0), off(1),
                               out-of-order(2) }
Kettle-state  ::= ENUMERATED { on(0), off(1),
                               out-of-order(2) }
lighter Lighter-state ::= on
kettle Kettle-state ::= lighter
```

Note the originality (or the ambiguity!) of this last example when the state of a lighter is used for defining the state of a kettle. This correspondence between compatibility of types and that of values have been introduced by three amendments called 'semantic model of ASN.1' (see Section 9.4 on page 121), which have been approved in June 1999. Some tools were applying such a correspondence before it was actually standardized.

To define a value set, it is given a name beginning with an upper-case letter, then comes the type (generally a name that begins with an upper-case letter) and finally after the symbol "`::=`", we give in curly brackets a series of values separated by the vertical bar "`|`" or a combination of the set operators defined in Section 15.5.2 on page 331:

```
PrimeNumbers INTEGER ::= { 2 | 3 | 5 | 7 | 11 | 13 }
```

Semantically, a value set is equivalent to a constrained type but its specific notation was introduced in 1994 to make it consistent with that of object sets. We shall come back on this very notion in Section 15.5 on page 329.

1st character of the first lexeme	**1st character of the second lexeme**	`::=`	**Assignment category**
`Upper-case-letter`		`::=`	type or information object class
`lower-case-letter`	`Upper-case-letter`[a]	`::=`	value or information object
`Upper-case-letter`	`Upper-case-letter`[a]	`::=`	value set or information object set

[a]From a merely lexical point of view [ASU86], a lexeme in upper-case letters cannot be decoded by a tool only as an information object class reference: in principle nothing prevents a whole type reference from being spelt in upper-case letters.

Table 9.1: Syntactical ambiguities in left parts of assignments

The definitions of information object classes, information objects and information object sets are discussed in greater length in Chapter 15.

Table 9.1 shows that if only the left-hand side of an assignment is being considered (before the "`::=`" symbol), the type, value and value set assignments are (respectively) similar to the class, object and object set assignments. This implies that tools like ASN.1 compilers should know the whole specification, using the `IMPORTS` clauses, to 'understand' it (i.e. to associate the right lexical token to each lexeme) since the ASN.1 syntax does not always permit to recognize an assignment category[1].

9.1.2 Reference Manual

$AssignmentList \rightarrow Assignment \ \cdots^{+}$

$Assignment \rightarrow TypeAssignment$
$\quad | \ ValueAssignment \quad | \ ValueSetTypeAssignment$
$\quad | \ ObjectClassAssignment \ | \ ObjectAssignment$
$\quad | \ ObjectSetAssignment \quad | \ \underline{ParameterizedAssignment}$

⟨1⟩ All the *Assignments* of a module must have distinct references (if two references differ from at least a letter case they are distinct).
⟨2⟩ If an *Assignment* reference also appears in the `IMPORTS` clause of a module, the rule ⟨22⟩ on page 116 applies.

[1]The problem gets worse when ASN.1 is used in an SDL specification, which is not (yet) case-sensitive.

TypeAssignment → typereference “::=” *Type*
Type → *BuiltinType*
  | *ReferencedType*
  | *ConstrainedType*

BuiltinType → *BitStringType* | *BooleanType*
  | *CharacterStringType* | *ChoiceType*
  | *EmbeddedPDVType* | *EnumeratedType*
  | *ExternalType* | *InstanceOfType*
  | *IntegerType* | *NullType*
  | *ObjectClassFieldType* | *ObjectIdentifierType*
  | *OctetStringType* | *RealType*
  | *RelativeOIDType* | *SequenceType*
  | *SequenceOfType* | *SetType*
  | *SetOfType* | *TaggedType*

ReferencedType → *DefinedType* | *UsefulType*
  | *SelectionType* | *TypeFromObject*
  | *ValueSetFromObjects*

⟨3⟩ Recursive types are allowed in ASN.1. The user should ensure that these types have at least one value of finite representation.

ValueAssignment → valuereference *Type* “::=” *Value*

⟨4⟩ *Value* must be a value expression of type *Type* or of a type that is compatible with *Type* according to the semantic model of ASN.1 (see Section 9.4 on page 121).

Value → *BuiltinValue*
  | *ReferencedValue*

⟨5⟩ Although recursive types are allowed in ASN.1 (see rule ⟨3⟩ on the current page), recursive values are forbidden.

BuiltinValue → *BitStringValue* | *BooleanValue*
  | *CharacterStringValue* | *ChoiceValue*
  | *EmbeddedPDVValue* | *EnumeratedValue*
  | *ExternalValue* | *InstanceOfValue*
  | *IntegerValue* | *NullValue*
  | *ObjectClassFieldValue* | *ObjectIdentifierValue*
  | *OctetStringValue* | *RealValue*
  | *RelativeOIDValue* | *SequenceValue*
  | *SequenceOfValue* | *SetValue*
  | *SetOfValue* | *TaggedValue*

ReferencedValue → *DefinedValue*
| *ValueFromObject*

TaggedValue → *Value*

ValueSetTypeAssignment → typereference *Type* "::=" *ValueSet*

⟨6⟩ Every value of the *ValueSet* must be of type *Type* or of a type that is compatible with *Type* according to the semantic model of ASN.1 (see Section 9.4 on page 121).
⟨7⟩ typereference denotes the subtype of *Type* that contains the values specified by *ValueSet* (rule ⟨11⟩ on page 333 draws a parallel between a value set and a subtype constraint).
⟨8⟩ A value set cannot be empty (see rule ⟨15⟩ on page 333).
⟨9⟩ If a value set should be renamed, the production *TypeAssignment* (see rule ⟨11⟩ on page 333) is to be used to give: `ValueSet2 ::= ValueSet1`.

ObjectClassAssignment → objectclassreference "::=" *ObjectClass*
ObjectAssignment →
objectreference *DefinedObjectClass* "::=" *Object*

⟨10⟩ The information object *Object* must be of class *DefinedObjectClass*.

ObjectSetAssignment →
objectsetreference *DefinedObjectClass* "::=" *ObjectSet*

⟨11⟩ Every information object in *ObjectSet* must be of class *DefinedObjectClass*.

9.2 Module structure

9.2.1 User's Guide

In ASN.1, a specification breaks down into one or several modules. The basic element for an ASN.1 compiler, the *module*, can be shared by several specifications, thereby facilitating their coherence. They contain mainly type assignments but also value, value set, class, object and object set assignments (even macro assignments if the module refers unfortunately to the ASN.1:1990 standard). These categories of assignments are presented in the next section.

Every module must comply with the *minimal* form:

```
ModuleName DEFINITIONS ::=
BEGIN
-- assignments
END
```

where `DEFINITIONS`, `BEGIN` and `END` are keywords and where the name of the module `ModuleName` respects the lexical rules of the token modulereference defined on page 102 (namely it starts with an upper-case letter).

It is preferable, as much as possible, to build up the module names with the number of the standard it refers to and its name or acronym. This would give for example a module called `ISO8823-PRESENTATION` in the standard [ISO8823-1] of the Presentation layer protocol (which we presented in Section 3.2 on page 20).

The module names may have a world-wide scope of action in which case it is obviously impossible to ensure uniqueness. In order to clear up this ambiguity, the module is registered in a world-wide registration tree. This procedure is described on page 163, where we deal with object identifiers. If the module is registered, its object identifier is inserted in curly brackets on the left-hand side of its name:

```
ModuleName {iso member-body(2) f(250) type-org(1)
  ft(16) asn1-book(9) chapter9(1) module1(0)}
DEFINITIONS ::=
BEGIN
-- assignments
END
```

If the specifier needs to invoke definitions from other modules they can be imported at the beginning of the module header with the `IMPORTS` clause (at the end of which we eventually find a singular semicolon: it is the only case, beside the symmetric `EXPORTS` clause, where it should be used in ASN.1):

```
ModuleName DEFINITIONS ::=
BEGIN
IMPORTS Type1  FROM Module1 {iso member-body(2) f(250)
                      type-org(1) ft(16) asn1-book(9)
                      chapter9(1) module1(0)}
        value2 FROM Module2 {iso member-body(2) f(250)
                      type-org(1) ft(16) asn1-book(9)
                      chapter9(1) module2(1)};
-- assignments
END
```

The object identifiers that follow `Module1` and `Module2` reference unambiguously the two modules (see on page 163).

It is *impossible* to import all the definitions of a module with a single notation, which could be "`IMPORTS ALL FROM Module1`". The specifier has to import every referenced assignment one by one. The references to parameterized assignments (see Chapter 17) can be followed by curly brackets:

```
ModuleName DEFINITIONS ::=
BEGIN
IMPORTS T{} FROM Module1;
U ::= T{INTEGER}
END
```

To indicate that a module does not use definitions from other modules the "`IMPORTS;`" clause can be inserted at the beginning of the body, after the keyword `BEGIN`.

The specifier may restrict the module interface by stating explicitly in an `EXPORTS` clause the assignments that can be imported by other modules:

```
ModuleName DEFINITIONS ::=
BEGIN
EXPORTS Type1, Type2;
IMPORTS Type1, value1 FROM Module1;

Type2 ::= SET { a Type1 DEFAULT value1,
                b BOOLEAN }
END
```

Note that the `EXPORTS` clause should necessarily appear before the `IMPORTS` clause. If a module does not contain the `EXPORTS` clause, all the definitions are visible from its interface. However, if it includes the "`EXPORTS;`" clause, none of its definitions can be imported by any other module (these are 'locked' in the module).

The notation "`EXPORTS ALL;`" will soon be added to the ASN.1 standard [ISO8824-1DTC2]. It will be equivalent to omitting the clause `EXPORTS`, i.e. all assignments defined (or imported) in the module can be referenced by some other module or imported in it.

In the header (i.e. after the keyword DEFINITIONS), a module may include two particular clauses:

```
ModuleName DEFINITIONS
  AUTOMATIC TAGS EXTENSIBILITY IMPLIED ::=
BEGIN
-- assignments
END
```

The clauses EXPLICIT TAGS, IMPLICIT TAGS or AUTOMATIC TAGS, which set the global tagging mode of the module, are presented in Section 12.1.3 on page 213. The clause EXTENSIBILITY IMPLIED, which makes all the CHOICE, SEQUENCE, SET and ENUMERATED types of the module extensible, is presented on page 252.

Finally, the specifier is free to format[2] the module as required and in particular, to start and finish the assignments anywhere in the specification. Indeed spaces, tabulations and newlines are meaningless for tools but for separating the tokens during the scanning stage (see Figure 22.1 on page 464).

9.2.2 Reference Manual

ModuleDefinition → *ModuleIdentifier* DEFINITIONS
TagDefault *ExtensionDefault* "::="
BEGIN *ModuleBody* END

⟨1⟩ The production *TagDefault* is defined on page 217.

ModuleIdentifier → modulereference *DefinitiveIdentifier*

⟨2⟩ The name of a module modulereference can be defined only once in the 'scope' of a specification (the standard does not define exactly what 'scope' is).
⟨3⟩ It is impossible to break down a module into several parts using each time the same modulereference.

DefinitiveIdentifier → "{" *DefinitiveObjectIdComponent* $\cdots^{+}$ "}"
| ϵ

⟨4⟩ The object identifier *DefinitiveIdentifier* uniquely and unambiguously identifies a module (see Section 10.8 on page 153).

[2] An ASN.1 mode for Emacs, which can be used on many operating systems, is available on the web site associated with this book (http://asn1.elibel.tm.fr/en/tools/emacs/). It offers a default formatting option but can be easily adapted to comply with every user's needs.

⟨5⟩ The object identifier *DefinitiveIdentifier* contains at least two arcs (*DefinitiveObjectIdComponent*). This restriction is imposed by the BER encoding rules, which encode together the two first arcs of the registration tree (see Section 18.2.8 on page 404).

$$
\begin{aligned}
\mathit{DefinitiveObjectIdComponent} \rightarrow\ & \mathit{NameForm} \\
\mid\ & \mathit{DefinitiveNumberForm} \\
\mid\ & \mathit{DefinitiveNameAndNumberForm}
\end{aligned}
$$

NameForm → identifier

DefinitiveNumberForm → number

DefinitiveNameAndNumberForm →
identifier "(" *DefinitiveNumberForm* ")"

⟨6⟩ identifier must be followed by a digit in round brackets (production *DefinitiveNameAndNumberForm*) except if it is one of the identifiers defined in Annexes A to C of the [ISO9834-1] standard, which are in yellow in Figure 10.4 on page 161.
⟨7⟩ The module object identifier (*DefinitiveIdentifier*) cannot contain a reference to a value (*DefinedValue*). ASN.1:1990 allowed the use of a reference to an `INTEGER` or `OBJECT IDENTIFIER` value in this case, but the compiler often imposed the *DefinedValue* to be defined in the identified module.

ExtensionDefault → `EXTENSIBILITY IMPLIED`
| ϵ

⟨8⟩ The clause `EXTENSIBILITY IMPLIED` imposes the automatic insertion of an extensible marker "..." at the end of all the `CHOICE`, `ENUMERATED`, `SEQUENCE` and `SET` types of the current module that do not explicitly contain such a marker. If this clause is not present in the module header, only the types including explicitly such a marker are extensible.
⟨9⟩ The clause `EXTENSIBILITY IMPLIED` only affects types: it has no effect on value sets, object sets or subtype constraints.
⟨10⟩ Before inserting a clause `EXTENSIBILITY IMPLIED` in the module header, the user should make sure that all `SEQUENCE`, `SET` and `CHOICE` types defined in this module respect the rule ⟨16⟩ on page 255 (particularly if these types include an extensible or untagged `CHOICE` type as one of their extensions).
⟨11⟩ To avoid checking the previous rule, the insertion of the clause `AUTOMATIC TAGS` in the header of the module should be preferred.

ModuleBody → *Exports Imports* $\underline{\mathit{AssignmentList}}$
| ϵ

Exports → `EXPORTS` *SymbolsExported* ";"
    | ϵ

SymbolsExported → *Symbol* "," $\cdots^*$

⟨12⟩ If the alternative ϵ is selected in the *Exports* rule (i.e. the keyword `EXPORTS` is not present), all the assignments defined (and imported) in the module are exported. This behavior was defined as the default case because the `EXPORTS` clause does not exist in the ASN.1:1984 standard.
⟨13⟩ The ASN.1 working group have agreed on allowing the clause "`EXPORTS ALL;`" in a module header. It will be equivalent to omitting the `EXPORTS` clause.
⟨14⟩ If no definition of the current module should be referenced in any other module, the clause "`EXPORTS;`" must be added in the module header.
⟨15⟩ Every reference (*Symbol*) in the `EXPORTS` clause is defined in the current module, otherwise it must appear only once in the `IMPORTS` clause of this module. The second part of the rule has been allowed since 1994 because this particular use appeared in some objects managed by GDMO and in ASN.1 modules related to network management (see Section 23.3 on page 482).

Imports → `IMPORTS` *SymbolsImported* ";"
    | ϵ

SymbolsImported → *SymbolsFromModule* $\cdots^*$

SymbolsFromModule →
  Symbol "," $\cdots^+$ `FROM` *GlobalModuleReference*

⟨16⟩ If the `EXPORTS` clause appears in the referenced module, every *Symbol* imported in the current module must appear in the *SymbolsExported* clause of the referenced module.
⟨17⟩ Every *Symbol* appearing in *SymbolsFromModule* must be defined in the body or in the `IMPORTS` clause of the referenced module (in the second case, *Symbol* can appear only once in the `IMPORTS` clause of the referenced module and should not be referenced in the body of that module).
⟨18⟩ For each *Symbol* referencing a type, the type is imported once tagged according to the global tagging mode (`EXPLICIT TAGS`, `IMPLICIT TAGS` or `AUTOMATIC TAGS`) of the module where it is defined. If the type is `ENUMERATED`, `SEQUENCE`, `SET` or `CHOICE` and if the module where it is defined includes the `EXTENSIBILITY IMPLIED` clause in its header, an extension marker "..." is inserted at the end of the definition if not present.

⟨19⟩ If a module contains an `IMPORTS` clause, the only external references (*ExternalTypeReference*, *ExternalValueReference*...) allowed in this module (see Section 9.3 on the next page) are those whose modulereference appears as a *GlobalModuleReference* in a *SymbolsFromModule* and whose (type, value...) reference appears in the *Symbols* list of the same *SymbolsFromModule*.
⟨20⟩ If no *Symbol* should be used in any external references (*ExternalTypeReference*, *ExternalValueReference*...), the clause "`IMPORTS;`" (followed only by a semicolon) must be inserted in the module header.
⟨21⟩ If there is no `IMPORTS` clause in the module header, this module can reference assignments from other modules using an external reference (*ExternalTypeReference*, *ExternalValueReference*...).
⟨22⟩ When a *Symbol* of the *SymbolsImported* list is also used to name an *Assignment* of the current module or when it appears several times in *SymbolsFromModule*s of the current module, it must be used only as an external reference (*ExternalTypeReference*, *ExternalValueReference*...) whose modulereference is the one that appears in the *GlobalModuleReference* following this *Symbol* (and not the one appearing in the header of the referenced module). An example will be given on page 164. It is not recommended to use this rule.

GlobalModuleReference → modulereference *AssignedIdentifier*

⟨23⟩ In *Imports*, all the modulereferences must be distinct and different from that of the current module.
⟨24⟩ In *Imports*, all the object identifiers *AssignedIdentifier* must be distinct and different from that of the current module.
⟨25⟩ If the object identifier *AssignedIdentifier* is present in *GlobalModuleReference*, it should necessarily be equivalent to (i.e. represent the same path in the registration tree as) the *DefinitiveIdentifier* that appears in the header *ModuleDefinition* of the referenced module.
⟨26⟩ *GlobalModuleReference* must include the same module name modulereference as in the header *ModuleDefinition* of the referenced module, except if *AssignedIdentifier* is present, in which case modulereference can be changed if necessary to prevent reference ambiguities (when the same *Symbol* is imported from two modules with the same modulereference, see rule ⟨22⟩ on this page).

$$
\begin{aligned}
\textit{AssignedIdentifier} \rightarrow\ & \underline{\textit{ObjectIdentifierValue}} \\
\mid\ & \underline{\textit{DefinedValue}} \\
\mid\ & \epsilon
\end{aligned}
$$

⟨27⟩ The object identifier *AssignedIdentifier* uniquely identifies the only module from which definitions can be imported (see on page 163).

⟨28⟩ *DefinedValue* is a reference to an `OBJECT IDENTIFIER` value. This alternative was not allowed in ASN.1:1990, i.e. *AssignedIdentifier* could not be a value reference.

⟨29⟩ Each *DefinedValue* appearing in *AssignedIdentifier* should:

- either be defined as an *Assignment* in the body of the current module and every valuereference that appears in the right-hand side of this *Assignment* (i.e. after "`::=`") must respect the current rule ⟨29⟩;
- or appear as *Symbol* in a production *SymbolsFromModule* of the current module with an object identifier *AssignedIdentifier* that contains no *DefinedValue*.

$$
\begin{aligned}
\textit{Symbol} \rightarrow\ & \textit{Reference} \\
\mid\ & \textit{ParameterizedReference} \\
\textit{Reference} \rightarrow\ & \mathsf{typereference} \mid \mathsf{valuereference} \\
\mid\ & \mathsf{objectclassreference} \mid \mathsf{objectreference} \\
\mid\ & \mathsf{objectsetreference} \\
\textit{ParameterizedReference} \rightarrow\ & \textit{Reference} \\
\mid\ & \textit{Reference}\ \text{“\{” “\}”}
\end{aligned}
$$

⟨30⟩ The two alternatives of the *ParameterizedReference* production are semantically equivalent. The second one should be preferred when the imported symbol is parameterized.

9.3 Local and external references

9.3.1 User's Guide

As in every computing language or formal notation, an ASN.1 definition is straightforwardly referenced by its name, which may start with an upper-case or lower-case letter (see Table 9.1 on page 108) according to the definitions of the lexical tokens given in Section 8.3 on page 100.

ASN.1 also offers the possibility of referencing an assignment that belongs to another module by means of an external reference. This reference consists of the assignment name preceded by the name of the module that exports it, and a dot as in:

```
MyType ::= SET OF OtherModule.Type
```

Even though it looks like a rather convenient notation, such an external reference is not recommended because it does not clearly isolate the interface from the module.

The `IMPORTS` clause should be preferred since it has the undeniable advantage of clearly indicating at the very beginning of a module, the other referenced modules. This clause also helps the programmer to quickly collect all the modules necessary for compiling the specifications (see Chapter 22).

Besides, if a module includes an `IMPORTS` clause, the module names appearing as a prefix of all external references used in the module should be mentioned in this clause. As a result, if a module has the clause "`IMPORTS;`" in its header, it cannot contain external references.

9.3.2 Reference Manual

Type reference (or value set reference)

DefinedType → *ExternalTypeReference*
| typereference
| *ParameterizedType*
| *ParameterizedValueSetType*

⟨1⟩ Productions *ParameterizedType* and *ParameterizedValueSetType* are defined on page 387.
⟨2⟩ typereference, *ParameterizedType* and *ParameterizedValueSetType* must name an *Assignment* of the current module or it must appear as a *Symbol* of the `IMPORTS` clause of the current module.

ExternalTypeReference → modulereference "." typereference

⟨3⟩ modulereference cannot reference the current module.
⟨4⟩ modulereference must appear only once in the `IMPORTS` clause of the current module (the rules ⟨22⟩ and ⟨26⟩ on page 116 can be used to find the referenced module thanks to the object identifier *AssignedIdentifier*), otherwise it must be the same modulereference as in the *ModuleIdentifier* appearing in the header of the referenced module.
⟨5⟩ If an `IMPORTS` clause appears in the header of the module referenced by modulereference, typereference can appear only once in this whole

clause. This rule does not clash with the rules ⟨22⟩ and ⟨26⟩ on page 116, which allow importing a *Symbol* in a single module `C` from two modules `A` and `B`; in this case this *Symbol* of module `C` cannot be referenced in a module `D` by an external reference (*ExternalTypeReference*, *ExternalValueReference*...).
⟨6⟩ It is recommended that the referenced module should include an `EXPORTS` clause and that typereference should appear in this clause.
⟨7⟩ typereference must name a *TypeAssignment* or *ParameterizedTypeAssignment*, or in a *ValueSetTypeAssignment* or *ParameterizedValueSetTypeAssignment* of the module referenced by modulereference, or else should appear as a *Symbol* in the `IMPORTS` clause of this referenced module.

Value reference

$$\begin{aligned} \textit{DefinedValue} \rightarrow\ & \textit{ExternalValueReference} \\ \mid\ & \mathsf{valuereference} \\ \mid\ & \underline{\textit{ParameterizedValue}} \end{aligned}$$

⟨8⟩ The production *ParameterizedValue* is defined on page 387.
⟨9⟩ valuereference and *ParameterizedValue* must name an *Assignment* of the current module or they must appear as a *Symbol* of the `IMPORTS` clause of the current module.

$$\textit{ExternalValueReference} \rightarrow \mathsf{modulereference}\ \text{“.”}\ \mathsf{valuereference}$$

⟨10⟩ Rules ⟨3⟩ and ⟨4⟩ on the preceding page also apply to modulereference.
⟨11⟩ If an `IMPORTS` clause appears in the header of the module referenced by modulereference, valuereference can appear only once in this whole clause (see rule ⟨5⟩ on the preceding page).
⟨12⟩ It is recommended that the referenced module should include an `EXPORTS` clause and that valuereference should be present in this clause.
⟨13⟩ valuereference must name a *ValueAssignment* of the module referenced by modulereference or it must appear as a *Symbol* in the `IMPORTS` clause of this referenced module.

Information object class reference

$$\begin{aligned} \textit{DefinedObjectClass} \rightarrow\ & \textit{ExternalObjectClassReference} \\ \mid\ & \mathsf{objectclassreference} \\ \mid\ & \underline{\textit{UsefulObjectClassReference}} \end{aligned}$$

⟨14⟩ objectclassreference must name an *ObjectClassAssignment* or *ParameterizedObjectClassAssignment* of the current module or it must appear as a *Symbol* of the `IMPORTS` clause of the current module.

ExternalObjectClassReference →
modulereference "." objectclassreference

⟨15⟩ Rules ⟨3⟩ and ⟨4⟩ on page 118 also apply to modulereference.
⟨16⟩ If an `IMPORTS` clause appears in the header of the module referenced by modulereference, objectclassreference can appear only once in this whole clause (see rule ⟨5⟩ on page 118).
⟨17⟩ It is recommended that the referenced module should include an `EXPORTS` clause and that objectclassreference appears in this clause.
⟨18⟩ objectclassreference must name an *ObjectClassAssignment* of the module referenced by modulereference or it must appear as a *Symbol* in the `IMPORTS` clause of this referenced module.

Information object reference

DefinedObject → *ExternalObjectReference*
| objectreference

⟨19⟩ objectreference must name an *ObjectAssignment* or *ParameterizedObjectAssignment* of the current module or it must appear as a *Symbol* of the `IMPORTS` clause of the current module.

ExternalObjectReference → modulereference "." objectreference

⟨20⟩ Rules ⟨3⟩ and ⟨4⟩ on page 118 also apply to modulereference.
⟨21⟩ If an `IMPORTS` clause appears in the header of the module referenced by modulereference, objectreference can appear only once in this whole clause (see rule ⟨5⟩ on page 118).
⟨22⟩ It is recommended that the referenced module should include an `EXPORTS` clause and that objectreference should appear in this clause.
⟨23⟩ objectreference must name an *ObjectAssignment* of the module referenced by modulereference or it must appear as a *Symbol* in the `IMPORTS` clause of this referenced module.

Information object set reference

DefinedObjectSet → *ExternalObjectSetReference*
| objectsetreference

⟨24⟩ objectsetreference must name an *ObjectSetAssignment* of the current module or it must appear as a *Symbol* in the `IMPORTS` clause of the current module.

ExternalObjectSetReference →
modulereference "." objectsetreference

⟨25⟩ Rules ⟨3⟩ and ⟨4⟩ on page 118 also apply to modulereference.
⟨26⟩ If an `IMPORTS` clause appears in the header of the module referenced by modulereference, objectsetreference can appear only once in this whole clause (see rule ⟨5⟩ on page 118).
⟨27⟩ It is recommended that the referenced module should include an `EXPORTS` clause and that objectsetreference should appear in this clause.
⟨28⟩ objectsetreference must name an *ObjectSetAssignment* or *ParameterizedObjectSetAssignment* of the module referenced by modulereference or it must appear as a *Symbol* in the `IMPORTS` clause of this referenced module.

9.4 The semantic model of ASN.1

The semantic model of ASN.1 aims at defining exactly what is semantically correct in ASN.1 to prevent tool programmers (see Chapter 22) from interpreting the standard differently. The bottom-line question is to clear up what is meant by the expression "*should be of the same type as*", which is frequently used in the standard and does not always tell exactly if the type can be tagged, constrained...

The history of the semantic model has been recounted from page 69 onwards. The three documents [ISO8824-1Amd2], [ISO8824-2Amd1] and [ISO8824-4Amd1] were approved by ITU-T in July 1999 and by ISO in February 2000.

By recursively applying the ordered list of transformations below [ISO8824-1Amd2, clause F.3.2], two expressions of types can be rewritten into two simpler but equivalent expressions, called *normalized*, whose syntaxes can be more easily compared:

1. remove the comments;
2. the following non-recursive transformations need only be applied once in any order :
   - associate a number with the identifiers without numbers in `ENUMERATED` types (see Section 10.4 on page 135), then re-order the *RootEnumeration*s in the alphabetical order of their identifiers;

- for `SEQUENCE` types:
  - cut any extension of the form "*ExtensionAndException ExtensionAdditions*" and paste it at the end of the *ComponentTypeLists*;
  - if `EXTENSIBILITY IMPLIED` is present in the module header, add an ellipsis "..." at the end of the *ComponentTypeLists*, if not yet present (see on page 252);
  - remove the *OptionalExtensionMarker* if present;
  - apply the `IMPLICIT TAGS` clause if it is present in the module header (see Section 12.1.3 on page 213);
  - if `AUTOMATIC TAGS` is present in the module header, decide whether automatic tagging must be applied (see rule ⟨6⟩ on page 223);
- for `SET` types:
  - cut any extension of the form "*ExtensionAndException ExtensionAdditions*" and paste it at the end of the *ComponentTypeLists*;
  - if `EXTENSIBILITY IMPLIED` is present in the module header, add an ellipsis "..." at the end of the *ComponentTypeLists*, if not yet present (see on page 252);
  - remove the *OptionalExtensionMarker* if present;
  - apply the `IMPLICIT TAGS` clause if it is present in the module header (see Section 12.1.3 on page 213);
  - if `AUTOMATIC TAGS` is present in the module header, decide whether automatic tagging must be applied (see rule ⟨6⟩ on page 227);
- for `CHOICE types`:
  - reorder the *RootAlternativeTypeList* in the alphabetical order of its identifiers;
  - if `EXTENSIBILITY IMPLIED` is present in the module header, add an ellipsis "..." at the end of the *ComponentTypeLists*, if not yet present (see on page 252);
  - remove the *OptionalExtensionMarker* if present;
  - apply the `IMPLICIT TAGS` clause if it is present in the module header (see Section 12.1.3 on page 213);

    - if `AUTOMATIC TAGS` is present in the module header, decide whether automatic tagging must be applied (see rule ⟨4⟩ on page 238);

  - the following transformations must be applied recursively in the specified order, until a fix-point is reached:

    (a) for each `OBJECT IDENTIFIER` value that begins with a *DefinedValue*, replace it with its definition;

    (b) develop the syntactic operators `COMPONENTS OF` (see Section 12.2 on page 218 and Section 12.3 on page 226);

    (c) develop the syntactic selection operator "<" (see Section 12.7 on page 239);

    (d) replace each *DefinedType* with its definition (see Section 9.1 on page 106) according to the following rules:

      - if the replacing type is a reference to the type being transformed, the *DefinedType* is replaced by a special lexical item that matches no other item than itself;
      - if the replaced type is a *ParameterizedType* or a *ParameterizedValueSetType*, replace every *DummyReference* with the corresponding *ActualParameter*;

    (e) replace all *DefinedValue* by its definition (see Section 9.1 on page 106); if it is a *ParameterizedValue*, replace every *DummyReference* with the corresponding *ActualParameter*;

3. move all constraints that apply to a `SEQUENCE OF` and `SET OF` type before the keyword `OF` (see Section 13.8 on page 275);

4. for each `SET` type, reorder the *RootComponentTypeList* in the alphabetical order of its identifiers;

5. for each `SEQUENCE`, `SET` and `CHOICE` for which it has been decided earlier to tag automatically, apply the automatic tagging (see Section 12.1.3 on page 213);

6. for each `INTEGER` value that is defined as an identifier, replace it by its associated number;

7. delete the *NamedNumberList*s that follow `INTEGER` types (as a consequence, two `INTEGER` types are compatible even if their named

number lists are not the same, but this rule does not apply to `ENUMERATED` types);

8. for each `BIT STRING` value that is defined as a list of identifiers, replace it by the corresponding bstring;

9. delete the *NamedBitList*s that follow `BIT STRING` types (as a consequence, two `BIT STRING` types are compatible even if their named bit lists are not the same);

10. suppress all trailing zero bits at the end of `BIT STRING` values (see rule ⟨15⟩ on page 150);

11. use systematically the numerical value (*NameForm*) for all the arcs of object identifiers of type `OBJECT IDENTIFIER` (see Section 10.8 on page 153) and `RELATIVE-OID` (see Section 10.9 on page 167);

12. remove spaces, tabulations and newlines at the beginning and at the end of bit strings, octet strings and character strings (see rules ⟨2⟩ on page 100, ⟨11⟩ on page 101 and ⟨8⟩ on page 101);

13. rewrite each character string of type `UTCTime` or `GeneralizedTime` so that it complies with the rules imposed by the DER encoding rules (see Table 19.2 on page 421);

14. modify each `REAL` value representation to get either an odd mantissa if the real is described in base 2, or a strictly-positive mantissa if the real is described in base 10 (by adapting the last digit of its mantissa as appropriate);

Having applied these rules, if the expressions of two normalized types are syntactically identical (term per term), the types are said 'compatible'. Otherwise, one must refer to the following rules:

- if a type is the subtype of another, they are compatible on their common set of values;

- two types involving different character string types can be compatible if a correspondence between the values of the character string types have been defined (see Section 11.14 on page 197);

- if the highest-level types have different tags, they may still be compatible; however, if two types `SEQUENCE`, `SET` or `CHOICE` are similar but for their respective tags, they are not compatible[3];
- no compatibility can exist (at least for the moment) between two information object classes or two information objects or two information object sets; hence two types that use one of these concepts in their definition cannot be compatible;

Again, two types are *compatible* means that any value of one type can be used every time a value of the other type is expected. This correspondence between values may occur whenever a value reference appears in an ASN.1 module, i.e. at the left-hand side of a value or information object assignment, after the `DEFAULT` clause, in a subtype constraint or in a value set specification.

Thus the types `T1` and `U1` below:

```
T1 ::= [0] SET { name   PrintableString,
                 age    INTEGER,
                 gender BOOLEAN -- male = TRUE -- }

U1 ::= [1] SET { gender Male,
                 name   VisibleString,
                 age    INTEGER }
Male ::= BOOLEAN
```

can be both normalized in (see Section 11.14 on page 197 for the character string type compatibility):

```
SET { age    INTEGER,
      name   UniversalString,
      gender BOOLEAN }
```

if we omit the higest-level types' tags. These types are compatible (with one another): a value of type `T1` can be used whenever a value of type `U1` is expected and vice versa (using character strings of type `PrintableString` for the component `name`). Other examples of compatibility between types were given on page 107.

[3]Such a rule, which somehow looks odd in the context of an ASN.1 semantic model, aims at simplifying the implementation of the model in a compiler. The few cases that are not considered are hardly ever used in practice: in particular, compatibility rules makes default values easier to define for the components of types `SEQUENCE` and `SET` (and these values are generally of type `INTEGER` or `OBJECT IDENTIFIER`).

However, the types T2 and U2 below:

```
T2 ::= [0] SEQUENCE { name [0] PrintableString,
                      age  INTEGER }

U2 ::= [1] SEQUENCE { name [1] PrintableString,
                      age  INTEGER }
```

are not compatible since their respective components `names` do not have the same tag.

These compatibility rules are fairly restrictive not to jeopardize the validity of existing specifications; nonetheless, it is recommended not to use type equivalence to keep specifications simple.

Note that the ASN.1 standard imposes no condition as to how, in practice, this semantic model should be implemented in the compilers.

Chapter 10

Basic types

Contents

Why is Lego the most ingenious toy in the world?

For a start, Sophie was not at all sure she agreed that it was. [...]

But she was a dutiful student. Rummaging on the top shelf of her closet, she found a bag full of Lego blocks of all shapes and sizes.

Jostein Gaarder, *Sophie's World.*

Without necessarily considering ASN.1 as the most ingenious toy in the world, we present here each and every one of its basic items. In the next chapter, we give the rules to comply with when combining these blocks to build up more complex types.

10.1 The `BOOLEAN` type

10.1.1 User's Guide

Let us start our journey in ASN.1's world by the simplest type to understand and manipulate: the boolean type, declared with the keyword `BOOLEAN`, whose two possible values are `TRUE` and `FALSE`.

At the end of a boxing round, the result can be modeled with the type:

```
RoundResult ::= BOOLEAN
```

and examples of results for a round are:

```
ok RoundResult ::= TRUE
ko RoundResult ::= FALSE
```

A type name describing the 'true state' is generally preferred to prevent any misunderstanding of the specification. For example, we call `Married` rather than `NotMarried`, the type that describes the marital status of a person:

```
Married ::= BOOLEAN
```

The `TRUE` value describes in a straightforward way a married person.

Likewise, it is better to call `Male` or `Female` rather than `Gender` (even though the latter is politically correct!) the boolean type that describes the gender of a person (thereby assuming that there are only two sorts). Therefore, we obtain:

```
kim Male ::= TRUE
```

for which we easily deduce Kim's gender.

Finally, to model several boolean values which convey a common concept (a group of check boxes in a graphical interface for example), we can use the `BIT STRING` type, which will be presented in Section 10.6 on page 145: it allows to emphasize the fact that the boolean values are linked with one another.

10.1.2 Reference Manual

Type Notation

BooleanType → `BOOLEAN`

⟨1⟩ This type has tag no. 1 of class `UNIVERSAL`.

⟨2⟩ The `BOOLEAN` type can be constrained by a single value (production

SingleValue on page 261) or by type inclusion (production *Contained-Subtype* on page 263), even though it is useless in practice.

Value notation

$$\begin{array}{lll} \textit{BooleanValue} & \rightarrow & \texttt{TRUE} \\ & \mid & \texttt{FALSE} \end{array}$$

10.2 The `NULL` type

10.2.1 User's Guide

Let us first mention an obvious paradox: as its name self-explicitly indicates, the type `NULL` models no information. It contains a single value, the value `NULL` (note the ambiguity between the value notation and the type notation), which conveys only one piece of information (when the value gets through, the receiver knows what semantic should be associated with it). As a result, the `NULL` type is the typical model for delivery reports or acknowledgements:

```
Ack ::= NULL
```

It is also used each time the information is not available because a value has not been given for transfer. Thus the model of a clock could be written:

```
Clock ::= CHOICE { time         UTCTime,
                   out-of-order NULL }
```

And when the clock battery is down, the value

```
battery-down Clock ::= out-of-order:NULL
```

is transmitted rather than return an inaccurate time or use a pre-defined value (24:00:00 for example) to stand for this state.

Neophytes not quite familiar with the `NULL` type tend to go for the `BOOLEAN` type instead and transmit (for example) the value `TRUE` for it. In addition to the fact that the `NULL` value has a more compact encoding than any other value, the use of a boolean type may lead to an erroneous interpretation of a specification since it implies that the reception of the `TRUE` value has a different meaning from that of the `FALSE` value.

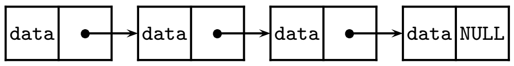


Figure 10.1: Linked list

One might try and use the `NULL` type to model an empty linked-list as programmers usually do, i.e. using the *null* end mark pointer (see Figure 10.1):

```
LinkedList ::= SEQUENCE {
  data Data,
  next CHOICE { linked-list LinkedList,
                end         NULL } }
```

But the built-in `SEQUENCE OF` (or `SET OF`) type, which will be presented in Section 12.4 on page 230, provides a model that is much simpler and gives a more compact encoding:

```
LinkedList ::= SEQUENCE OF Data
```

10.2.2 Reference Manual

Type notation

NullType → `NULL`

⟨1⟩ This type has tag no. 5 of class `UNIVERSAL`.
⟨2⟩ The `NULL` type can be constrained by a single value (production *SingleValue* on page 261) or by type inclusion (production *ContainedSubtype* on page 263), even though it is useless in practice.

Value notation

NullValue → `NULL`

10.3 The `INTEGER` type

10.3.1 User's Guide

The integers can be declared in ASN.1 by the keyword `INTEGER`, which stand for any positive or negative integer whatever its length[1], namely

[1]Note, however, that this set of values does not include singular values like $+\infty$ or $-\infty$. We shall see on page 140 that these values are members of the `REAL` type.

the set **Z** in mathematics:

```
zero INTEGER ::= 0
french-population INTEGER ::= 60000000
fridge-temperature INTEGER ::= -18 -- in Centigrade scale
```

If transmitting great numbers is necessary, a representation that could limit the encoding and enable their effective use by the communicating applications should be agreed on: indeed, the system on which these applications are running and the languages in which they are implemented are very unlikely to manage such numbers easily.

This generality of its `INTEGER` type is sometimes held against ASN.1 though it is always better to 'have' than 'have not'; one just needs to restrict this integer type to an interval using a subtype constraint:

```
Interval ::= INTEGER (123456789..123456790)
```

to make out the problem. In this case, the decoder knows beforehand the size of the data to be received and can allocate the corresponding memory needed for storing it, provided the ASN.1 compiler took into account the subtype constraints when generating the decoder. Moreover, if the packed encoding rules (PER) are used, the size of these data can be notably cut down compared to a BER encoding. For example, a value of the type `Interval` above is encoded with the PER on... 1 bit! (see Chapter 20.)

In some cases (for the identification of error codes in a specification for instance), it can be interesting to give a particular name to some values in order to make them easier to understand and improve the interface between the communicating applications and the encoders and decoders. Such information can, of course, be given in comments in the specification but these will not be used by the compiler since it systematically ignores them at the earliest step of the lexical-analysis stage [ASU86].

ASN.1, therefore, provides a specific syntax for the `INTEGER` type: the list of integers, where every one of them is preceded by its identifier (which begins with a lower-case letter); this list is denoted in curly brackets[2] after the keyword `INTEGER`.

[2]The reader will be easily convinced by subsequent sections of the Reference Manual that the keys "{" and "}" must have been the most 'close-at-hand' for the ASN.1 designers, for these symbols are used in the standard to denote completely different concepts! We will see on page 469 that this peculiarity can be tricky to deal with when programming tools for ASN.1.

The error code of a floppy can then be modeled with the type:

```
ErrorCode ::= INTEGER { disk-full(1), no-disk(-1),
    disk-not-formatted(2) }
stupid-error ErrorCode ::= disk-full
```

these identifiers have only a local scope, i.e. they can only be used to define values of type `ErrorCode` like `stupid-error` for example. The same number can obviously not be named by two different identifiers and a given identifier cannot name two different numbers. The named integers are not necessarily ordered or consecutive and the list of integers in curly brackets is not restrictive[3].

The integers that are not named remain accessible, so that non-explicitly listed errors can be specified:

```
ok ErrorCode ::= 0
```

We can also write:

```
stupid-error ErrorCode ::= 1    -- disk full
```

even though the use of an identifier is much recommended when the integer involved is named.

The type `INTEGER` is often used to model an error code, as in the Presentation layer protocol [ISO8823-1] for example:

```
Abort-reason ::= INTEGER {
  reason-not-specified(0),
  unrecognized-ppdu(1),
  unexpected-ppdu(2),
  unexpected-session-service-primitive(3),
  unrecognized-ppdu-parameter(4),
  unexpected-ppdu-parameter(5),
  invalid-ppdu-parameter-value(6) }
```

Another classical use is that which consists in naming particular integers to give them a specific meaning:

```
Temperature ::= INTEGER { freezing(0), boiling(100) }
  -- in Centigrade scale
```

Contrary to the `ENUMERATED` type, the list of identifiers cannot be numbered automatically; the integers must be explicitly associated with their identifiers since it is obviously these integers that are encoded for transmission. The type `INTEGER` is not extensible either: the extension

[3]See the difference with the `ENUMERATED` type on page 135.

marker "..." cannot be used (except if the type has an extensible subtype constraint, see on page 291), but this is no problem since, contrary to the `ENUMERATED` type, the list of named integers is not restrictive.

Even though this is no common usage, numbers in round brackets can be references to integer values defined somewhere else in the module or imported from some other module. If these identifiers and value references are not properly chosen, this can result in a rather obscure specification as in:

```
alpha INTEGER ::= 1
Type1 ::= INTEGER { alpha(2) }
Type2 ::= INTEGER { alpha(3), beta(alpha) }
gamma Type2 ::= beta
delta Type2 ::= alpha
```

In `Type2`, the word '`alpha`' in round brackets is necessarily a value reference and should be therefore represented by the value `1` (see rule ⟨10⟩ on the following page). For the value `gamma`, the word '`beta`' is first looked for in the scope of the `INTEGER` type `Type2`; so `gamma` equals `1`. For the value `delta`, the word '`alpha`' is first looked for in the scope of the type `Type2`, so `delta` equals `3`.

No compatibility is defined between the types `INTEGER` and `REAL` (see Section 9.4 on page 121), that is to say that an `INTEGER` value cannot be allocated to a value of type `REAL`:

```
integer INTEGER ::= 5
real REAL ::= integer   -- forbidden
```

Likewise, `INTEGER` and `ENUMERATED` types are not compatible either. But two `INTEGER` types are compatible whatever the named number list that can be associated to them may be.

10.3.2 Reference Manual

Type notation

IntegerType → `INTEGER`
| `INTEGER` "{" *NamedNumber* "," $\cdots^{+}$ "}"

⟨1⟩ This type has tag no. 2 of class `UNIVERSAL`.
⟨2⟩ For the second alternative, the type is not restricted to the set of integers in curly brackets.
⟨3⟩ The type `INTEGER` can be constrained by a single value (production

SingleValue on page 261), by type inclusion (production *Contained-Subtype* on page 263) or by an interval (production *ValueRange* on page 265).
⟨4⟩ Two `INTEGER` types are compatible according to the semantic model of ASN.1 (see Section 9.4 on page 121) even if their associated named number lists differ.
⟨5⟩ When an `INTEGER` type with a named integer list is imported into another module, the identifiers in this list are imported too. These can only be used to define a value of this type.

NamedNumber → identifier "(" *SignedNumber* ")"
| identifier "(" *DefinedValue* ")"

⟨6⟩ The numbers' ordering is not significant.
⟨7⟩ The identifiers in curly brackets must be distinct.
⟨8⟩ The integers defined, explicitly (*SignedNumber*) or by reference (*DefinedValue*), in curly brackets must be distinct.
⟨9⟩ *DefinedValue* must be a reference to a value of type `INTEGER`.
⟨10⟩ The lexeme representing the *DefinedValue* cannot be interpreted as one of the identifiers defined in the `INTEGER` type since an identifier cannot be used in round brackets.

Value notation

IntegerValue → *SignedNumber*
| identifier

⟨11⟩ The identifier must be one of those appearing in the corresponding *IntegerType*.
⟨12⟩ When defining an integer value, if the allocated value is associated with an identifier in the type definition, it is recommended to use this identifier instead.
⟨13⟩ If identifier does not appear in the named number list defined by the associated *IntegerType*, identifier must be interpreted as a valuereference in a *DefinedValue*, i.e. a reference to a value of type `INTEGER`.

10.4 The `ENUMERATED` type

10.4.1 User's Guide

Sometimes confused with the type `INTEGER`, the type of enumerations is declared with the keyword `ENUMERATED`, and should be distinguished from the former for the following reasons:

- the enumeration is restrictive, i.e. the numbers that are not listed in curly brackets do not belong to the type;
- semantically, the named integers are not numbers, so that they cannot be manipulated by operators[4] but they are only used for encoding;
- integers do not have to be explictly associated with the identifiers since these can automatically be computed by an ASN.1 compiler;
- the enumeration can be extended if a new version of the module is produced to ensure the compatibility of the encoding;
- to sum it up: for an `ENUMERATED` type, a number is associated (implicitely or explicitely) with every identifier whereas, for an `INTEGER` type, an identifier is (explicitely) associated with each integer.

The `ENUMERATED` type, therefore, is used each time we should inventory objects[5]. It is frequently chosen to describe the states of a system or an error report as in the ACSE standard for the association control service element [ISO8650-1]:

```
ABRT-diagnostic ::= ENUMERATED {
  no-reason-given(1), protocol-error(2),
  authentication-mechanism-name-not-recognized(3),
  authentication-mechanism-name-required(4),
  authentication-failure(5),
  authentication-required(6),
  ...  }
```

[4]To confirm this, one can note, for example, that the SDL formalization language (see Annex B on page 509) provides no operators to manipulate enumerated values.

[5]"*Everything can be listed: the editions of Tasso, the islands on the Atlantic Coast, the ingredients required to make a pear tart, the relics of the major saints, masculine substantives with a feminine plural (*amours, délices, orgues*), Wimbledon finalists* [...]" (Georges Perec, *Think/Classify, The ineffable joys of enumeration*, translated by John Sturrock, in *Species of Spaces and Other Pieces*).

The ENUMERATED type can also[6] model the radio-buttons of an MMI window:

```
RadioButton ::= ENUMERATED { button1(0), button2(1),
                  button3(2) }
selected-by-default RadioButton ::= button1
selected-by-default RadioButton ::= 0       -- forbidden
```

The name of the type, RadioButton, is grammatically singular since only one button can be selected at a time. Note that contrary to the INTEGER type, the identifier must be used when defining a value like selected-by-default.

The same readability problems as those evoked on page 133 for the INTEGER type where both the value and the identifier were called alpha also apply to the ENUMERATED type. Still, for there is no compatibility defined between the ENUMERATED type and the INTEGER type, a value of type ENUMERATED cannot be used to define a value of type INTEGER (and reverse). Two ENUMERATED types cannot be compatible either.

As mentioned already, contrary to the INTEGER type, the integers listed after an ENUMERATED type are useful only for encoding. A manual numbering of the states, especially if this only consists in a trivial increment may seem useless. This is the reason why ASN.1 provides an automatic numbering for the identifiers (with an initial counter value 0). Thus the type:

```
RadioButton ::= ENUMERATED { button1, button2, button3 }
```

is equivalent to the type RadioButton previously defined. We should keep in mind that the automatic numbering can annihilate the encoding compatibility among ENUMERATED types if those are extended by insertion of new identifiers at the beginning or in the middle of their enumeration list (the numbers are then accordingly shifted). This is why the notion of extensibility has been introduced since 1994[7].

When writing a specification, we can indicate that the ENUMERATED type can be extended in subsequent versions of this specification by inserting an extension marker "..." at the end of it. As a result, the following type RadioButton contrary to that which was defined earlier,

[6]Other examples of ENUMERATED types are presented in [ISO8824-1, clause C.2.3].

[7]We shall come back to the notion of extensibility in Section 12.9 on page 244 when we describe the types CHOICE, SEQUENCE and SET.

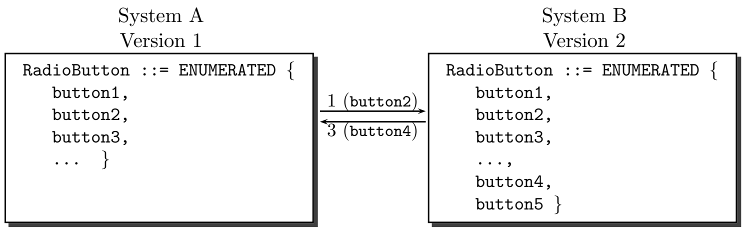


Figure 10.2: Data exchange between two systems whose specification versions are different

is declared extensible in the first version of the specification:

```
RadioButton ::= ENUMERATED { button1, button2, button3,
                             ... }
```

and may be rewritten and extended as:

```
RadioButton ::= ENUMERATED { button1, button2, button3,
                             ..., button4, button5 }
```

in a second version.

An ASN.1 compiler would then generate an encoder and a decoder that could manage a communication between two systems whose respective protocol versions are different. Figure 10.2 shows that if System A sends the value `1` for `button2`, System B can decode it since it knows (at least) the version 1 of the specification. If System B sends the value `3` for `button4`, System A does not necessarily know how to decode this value, which was defined in the version 2 of the specification, but will recover (discarding the value) and carry on decoding the byte streams of the data.

The type `ABRT-diagnostic` on page 135 is an example of an extensible `ENUMERATED` type.

A few extra rules on automatic numbering are necessary when the `ENUMERATED` type is extensible (these are described in the next 'Reference Manual' section). In particular, the integers after the extension markers must be listed in increasing order. The following examples illustrate this:

```
A ::= ENUMERATED { a, b, ..., c(0) }
      -- Impossible:  a and c equal 0
B ::= ENUMERATED { a, b, ..., c, d(2) }
      -- Impossible:  c and d equal 2 (see rule ⟨10⟩)
C ::= ENUMERATED { a, b(3), ..., c(1) }
      -- Correct:  c = 1
```

	BER	**PER**
`ChoiceOfNull`	2 bytes	3 bits
`Enumeration`	3 bytes	3 bits

Table 10.1: Comparison between the BER and PER encodings of an enumeration and a choice of `NULL` types

```
D ::= ENUMERATED { a, b, ..., c(2) }
      -- Correct:  c = 2
E ::= ENUMERATED { a, b, ..., c }          -- c = 2
F ::= ENUMERATED { a, b, c(0), ..., d }    -- d = 3
G ::= ENUMERATED { a, b, ..., c(3), d }
      -- d = 4 (see rule ⟨10⟩ on page 140)
```

Contrary to the constructed types `CHOICE`, `SEQUENCE` and `SET`, the type `ENUMERATED` includes only one extension marker (i.e. new identifiers must be included at the end of an enumerated list without the version double square brackets "`[[`" and "`]]`").

An exception marker "`!`" cannot be associated with the extension marker "`...`" as the notation stands at present, but this will be changed in the near future.

In a specification dealing with a multimedia protocol (with PER encoding, see Chapter 20), types with the following syntax can occasionally be found[8]:

```
ChoiceOfNull ::= CHOICE { e1 NULL,
                          e2 NULL,
                          e3 NULL,
                          e4 NULL,
                          e5 NULL,
                          e6 NULL }
```

although one might have preferred:

```
Enumeration ::= ENUMERATED { e1(1), e2(2), e3(3), e4(4),
                             e5(5), e6(6) }
```

Table 10.1 shows that although the type `ChoiceOfNull` is not as easily read and interpreted in that case, its PER encoding has the same length (3 bits) as the type `Enumeration`.

[8]This type is supposed to be defined in a module which includes the `AUTOMATIC TAGS` clause in its header.

10.4.2 Reference Manual

Type notation

EnumeratedType → `ENUMERATED` "{" *Enumerations* "}"

⟨1⟩ This type has tag no. 10 of class `UNIVERSAL`.
⟨2⟩ The type `ENUMERATED` can be constrained by a single value (production *SingleValue* on page 261) and by type inclusion (production *ContainedSubtype* on page 263).
⟨3⟩ Two `ENUMERATED` types cannot be compatible according to the semantic model of ASN.1 (see Section 9.4 on page 121) if their named number lists are different (i.e. with different identifiers or different numbers).
⟨4⟩ When an `ENUMERATED` type with a named number list is imported into another module, the identifiers in this list are imported too. These can only be used to define a value of this type.

Enumerations → *RootEnumeration*
| *RootEnumeration* "," "..." *ExceptionSpec*
| *RootEnumeration* ","
"..." *ExceptionSpec* ","
AdditionalEnumeration

⟨5⟩ The extension marker "..." was added in 1994.
⟨6⟩ Contrary to the extensible types `CHOICE`, `SEQUENCE` and `SET`, the version square brackets "`[[`" and "`]]`", and the second extension marker "..." are not allowed in *Enumerations*.

In the near future [ISO8824-1DTC2], an exception marker "`!`" will be allowed with the extension marker "..." to let the application know that the received value conforms neither to the extension root nor to the extensions so that this exception could be handled by a bespoke routine (see on page 247).

RootEnumeration → *Enumeration*

⟨7⟩ In the *RootEnumeration*, the integers are neither necessarily ordered, nor necessarily successive.

AdditionalEnumeration → *Enumeration*

⟨8⟩ In *AdditionalEnumeration* (i.e. after the extension marker "..."), the integers must be ordered but not necessarily successive.
⟨9⟩ In *AdditionalEnumeration*, every explicit value (*SignedNumber*) or referenced value (*DefinedValue*) must be different from all the explicit,

referenced or calculated values (see rule ⟨12⟩ on the current page) that precede it in the *RootEnumeration*.
⟨10⟩ In *AdditionalEnumeration*, every explicit, referenced or calculated value must be greater than any of the values (explicit, referenced or calculated) that precede it syntactically in *AdditionalEnumeration*.

Enumeration → *EnumerationItem* "," $\cdots^{+}$
EnumerationItem → identifier
| *NamedNumber*

⟨11⟩ The identifiers can be enumerated without systematically associating an integer to them explicitly.
⟨12⟩ Each identifier that is not followed by a value in round brackets is assigned an integer, by one-increment starting from 0 but excluding those (explicit, referenced or calculated) already used in the *NamedNumbers* that precede it syntactically. The extension marker "..." does not affect this procedure for allocating integers.

NamedNumber → identifier "(" *SignedNumber* ")"
| identifier "(" *DefinedValue* ")"

⟨13⟩ In *Enumerations*, the identifiers must be distinct.
⟨14⟩ *DefinedValue* should be of type `INTEGER`.
⟨15⟩ The lexeme *DefinedValue* cannot be interpreted as an identifier defined in the `ENUMERATED` type since an identifier cannot appear in round brackets.

Value notation

EnumeratedValue → identifier

⟨16⟩ identifier must be one of those appearing in the associated *EnumeratedType*.
⟨17⟩ The existence of an extension marker "..." in the associated *EnumeratedType* does not alter the previous rule.

10.5 The `REAL` type

10.5.1 User's Guide

The real numbers are just like the other real numbers in information technology: they should be called decimals! The type `REAL` in ASN.1

can model arbitrarily long but finite decimals. There is nothing preventing us from using the REAL type to transmit the first ten thousand decimals of π! The transmission of numbers with an infinite decimal part like $\frac{1}{3}$, π or e, however, would require an appropriate if not dedicated model. From the specifier's viewpoint, a definition such as the following is perfectly fine[9]:

```
ExtendedReal ::= CHOICE {
  decimal           REAL,
  particular-real   ENUMERATED {one-third, pi, e, ...} }
```

where it is up to the communicating applications to use their own representation for the real numbers with an infinite decimal part. In practice, the REAL type is hardly ever used in ASN.1 specifications, mostly because it does not provide the well-known notation with a dot like 3.14 (but this is going to be amended soon as we will see on page 143).

In fact, all the real numbers in ASN.1 can be written $m \times b^e$ where the mantissa m is of type INTEGER, the base b equals 2 or 10 and the exponent e is also a number of type INTEGER. We have for example:

```
pi REAL ::= { mantissa 314159, base 10, exponent -5 }
e REAL ::= { mantissa 271828128459045235360287,
             base 10,
             exponent -23 }
zero REAL ::= 0
```

The identifiers mantissa, base and exponent must be used in the definition of real values since the 1994 standard. Indeed, in ASN.1:1994, the REAL type has been defined as if it were semantically equivalent to the type:

```
SEQUENCE { mantissa INTEGER (ALL EXCEPT 0),
           base     INTEGER (2|10),
           exponent INTEGER }
```

but with a specific tag of class UNIVERSAL[10]. It makes it possible for an encoding different from that of the generic SEQUENCE type to be associated with it.

[9]We will see in Chapter 18 that encoding rules for the REAL type may theoretically be extended to include these particular real values (the 'real' difficulty would then be to decide which of these values should be kept for the standard).

[10]The tags of class UNIVERSAL are defined in Table 12.1 on page 209.

In ASN.1:1990 (the `REAL` type did not exist in ASN.1:1984), the three values above mentioned were written:

```
pi REAL ::= { 314159, 10, -5 }
e REAL ::= { 27182812845904523536O287, 10, -23 }
zero REAL ::= 0
```

The real value zero cannot be represented by means of a 3-tuple ⟨mantissa, base, exponent⟩ where the mantissa would equal zero (i.e. the real `0` of ASN.1 is neither in base `2`, nor in base `10`). The necessity[11] of denoting it `0` can be taken advantage of to associate a specific encoding with this value. A `REAL` value can also be defined with the keywords `PLUS-INFINITY` or `MINUS-INFINITY` to denote the particular values $+\infty$ or $-\infty$ (contrary to the `INTEGER` type, any value from the whole range of the decimal numbers can be modeled).

Apart from the three previous values, the type `INTEGER (2|10)` of the `base` component states that ASN.1 real numbers can be represented[12] either in base 2, which is similar to the processor internal storage of floats, or in base 10 for a more classical representation.

Note, however, that the two abstract values {`mantissa 5, base 2, exponent 0`} and {`mantissa 5, base 10, exponent 0`}, though denoting the same real number `5`, cannot be considered as having the same semantics from the communicating application's viewpoint (we will see in Part III debating the encoding rules and beginning on page 391, that these two values' encoding are different, even with canonical encoding rules like DER or canonical PER).

If this representation of the real numbers in 3-tuples is judged awkward to use, it is possible to provide real numbers to the encoder via an MMI similar to the one which was designed by the X/Open [TMF96] consortium, using one of the following representations of the [ISO6093] standard:

- NR1: "`3`", "`-1`", "`+1000`";
- NR2: "`3.0`", "`-1.3`", "`-.3`";
- NR3: "`3.0E1`", "`123E+100`".

[11] The lexeme `0` should not be confused with the zero value of type `INTEGER` since the type that governs an ASN.1 value is always known.

[12] If a number has a finite representation in base 2, it has a finite representation in base 10, but the reverse is not correct; see, for example, the following counter-example: $0.2_{10} = 0.00110011..._2 = 0.[0011]_2$.

The NR1 (Numerical Representation 1) is the fixed decimal point representation: the decimal place is implicit and predefined. The NR2 format is the non-graded representation with explicit decimal point: it corresponds to the usual writing using the dot or the comma as the decimal separator. The NR3 format is the graded representation with explicit decimal point: it includes a signed exponent (base 10) that may be followed by an optional letter `e` or `E`.

The ASN.1 working group have considered the possibility of allowing the usual notation that uses a decimal point for defining `REAL` values. If this proposition is accepted [ISO8824-1DTC3], it will be possible to write for example:

```
pi REAL ::= 3.14
```

Since 1994 (the year when the `REAL` type was semantically defined with a `SEQUENCE` type, see on page 141), it has been possible to constrain `REAL` types very precisely to make communicating systems interwork more easily. It allows to keep only the binary representations that are the closest to floating-point processors (see Section 18.2.5 on page 400) and are thus the most common:

```
BinaryReal ::= REAL (WITH COMPONENTS {..., base (2)})
```

or even restrict all the three components at once:

```
RestrictedReal ::= REAL (WITH COMPONENTS {
    mantissa (-16777215..16777215),
    base     (2),
    exponent (-125..128) })
```

Finally, as mentioned already while introducing the `INTEGER` type, an `INTEGER` value cannot be used to define a `REAL` value. The declaration of the value `real` below is therefore erroneous:

```
integer INTEGER ::= 5
real REAL ::= integer -- forbidden
```

10.5.2 Reference Manual

Type notation

RealType → `REAL`

⟨1⟩ This type has tag no. 9 of class `UNIVERSAL`.

⟨2⟩ Since 1994, the `REAL` type has been semantically equivalent to the type:

```
[UNIVERSAL 9] IMPLICIT SEQUENCE {
  mantissa INTEGER (ALL EXCEPT 0),
  base     INTEGER (2|10),
  exponent INTEGER }
```

⟨3⟩ The `REAL` type can be constrained by a single value (production *SingleValue* on page 261), by type inclusion (production *ContainedSubtype* on page 263), by interval (production *ValueRange* on page 265) and, since 1994, by constraints on the components of the `SEQUENCE` type (production *InnerTypeConstraints* on page 277). Examples of subtyping have been given in the 'User's Guide' section above.
⟨4⟩ If a `REAL` type is contrained by *InnerTypeConstraints* (see on page 277), this subtype constraint does not apply on the particular values `0`, `MINUS-INFINITY` and `PLUS-INFINITY` (the operator `EXCEPT` on page 290 must be used to forbid one of them).

Value notation

RealValue → *NumericRealValue*
  | *SpecialRealValue*

NumericRealValue → `0`
  | *SequenceValue*

⟨5⟩ *SequenceValue* cannot represent the value `0` because the value `0` in ASN.1 is neither in base `2` nor in base `10`.
⟨6⟩ *SequenceValue* must be a value conforming to the `SEQUENCE` type associated with the `REAL` type and defined in rule ⟨2⟩ on this page.
⟨7⟩ The name of the components `mantissa`, `base` and `exponent` (see rule ⟨2⟩ on the current page) did not exist in ASN.1:1990 and a `REAL` value was written '`{ m, b, e }`'.
⟨8⟩ A communicating application can associate two different semantics with two different *RealValue*s that denote the same real number when one of those is in `base 2` while the other is in `base 10`.

SpecialRealValue → `PLUS-INFINITY`
  | `MINUS-INFINITY`

10.6 The BIT STRING type

10.6.1 User's Guide

In order to comply with the *esprit libertaire* of ASN.1, the binary string can be of null length... or arbitrarily long. The `BIT STRING` type (two words, no dash) is used to transmit data that are inherently binary (a compressed facsimile or encrypted data, for example) or to model boolean vectors (a list of choices in an MMI window as in Figure 10.3 on the following page). The `BIT STRING` type should be used only when it is absolutely necessary.

In the [X.509] standard of the directory service, we find the type:

```
SubjectPublicKeyInfo ::= SEQUENCE {
  algorithm        AlgorithmIdentifier,
  subjectPublicKey BIT STRING }
```

to transmit identification public key and the corresponding encryption algorithm.

A binary string is presented in quotes and followed by the capital letter `B` if it is made of binary digits (`0` or `1`) or the capital letter `H` if these are hexadecimal digits (from `0` to `9` and from `A` to `F` in upper-case letters only). Since 1994, if a long string should appear as an abstract value in an ASN.1 module, it can be split across lines and may therefore include tabulations, newlines and spaces (which, of course, are meaningless):

```
pi-decimals BIT STRING ::=
   '00100100001111110110101010001000100001
    01101000110000100011010011000100110001
    10011000101000101110000000110111000'B
pi-decimals BIT STRING ::=
   '243F6A8885A308D313198A2E0370'H
```

Concerning the ordering of the bits, the following convention was adopted: the *first* bit of the string is the one on the left-hand side (position 0) and the *last* bit is the one on the right-hand side. Of course, this convention has no implication whatsoever on the ordering of the bits handled by the communicating applications or by the communication medium. Indeed, "*'all true believers shall break their eggs at the convenient end': and which is the convenient end, seems, in my humble opinion, to be left to every man's conscience, or at least, in the power of the chief magistrate to determine*". This quote of Swift's *Gulliver's travels* should remind the reader of our discussion on page 8, concerning the agreement on the bit orders.

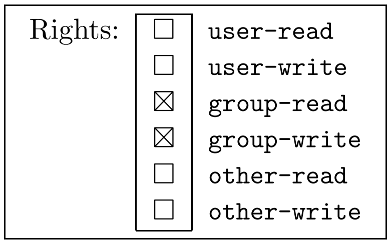


Figure 10.3: Block of check boxes

The size of the binary strings should be limited whenever possible using a subtype constraint so that the encoder and the decoder should allocate the adequate number of blocks but also make the encoding more compact:

```
StringOf32Bits ::= BIT STRING (SIZE (32))
```

ASN.1 offers a particular syntax for the `BIT STRING` type to declare special positions of bits using round brackets. The value definition for such types consists in listing in curly brackets (again!) the position of the bits set to `1` (the positions that are not mentioned or not at the end of the string take the default value `0`, and the value '{}' represents the empty binary string `''B` or `''H`). This syntax can easily model the rights attributed to a file in the *UNIX* operating system (reading and writing for the owner, for the owner's group and for the other users) whose representation is given on Figure 10.3:

```
Rights ::= BIT STRING { user-read(0), user-write(1),
                        group-read(2), group-write(3),
                        other-read(4), other-write(5) }
group1 Rights ::= { group-read, group-write }
```

These values can always be defined as bit or hexadecimal strings; in this case, positions other than those named in the type can have the value `1` as illustrated with the value `weird-rights`:

```
group2 Rights ::= '0011'B
group2 Rights ::= '3'H
group3 Rights ::= '001100'B
weird-rights Rights ::= '0000001'B
```

These examples call for three remarks concerning the definition of the type `Rights`:

- the list of named bits does not influence the set of abstract values of this type;

- the list of named bits is not restrictive (the positions are actually not necessarily consecutive[13]); no size constraint is implicit as seen in the value `weird-rights` on the preceding page;

- the trailing zero bits are not significant[14] and the value `group2` and `group3` represent the same value as `group1`; the zero bits at the beginning of the string are obviously compulsory so that the bits `1` could properly be positioned.

Because the named bit list implies no constraint on the size of the type, the application should be able to receive a value whatever the number of zeros added at the end (for example, `'001100000'B` as a value for the type `Rights`). Canonical encoding rules like DER or canonical PER, remove these bits to ensure canonicity (only if the type has a list of named bits, of course).

This convention can be easily extended to the 'named bit list' case if the trailing zeros are agreed on to correspond to the status quo: the values that conformed to the old version remain valid for the new one. Note, however, the reverse problem: if the zero bits have to remain significant in the next version of the protocol that is being modeled, it becomes impossible to associate a list of named bits with the `BIT STRING` type.

Contrary to the `ENUMERATED` type, all identifiers of the bit positions should have an associated number because these positions are not necessarily consecutive so that there can be no automatic numbering procedure by default. Besides, the list of the named positions cannot be declared as extensible using the extension marker "...".

If these identifiers are not properly chosen, the specification can be more difficult to read as in the following example:

```
alpha INTEGER ::= 1
BinaryString ::= BIT STRING { alpha(3), beta(alpha) }
```

[13]This is the case, for example, for the type:

```
ShinyDays ::= BIT STRING { first(1), last(31) }
```

where the positions of `first` and `last` have been specified only to make the meaning of this type more explicit, but do not limit the size of the strings to 32 bits (including the bit in position `0`).

[14]The constraint on these bits has been relaxed since 1988 to solve interworking problems encountered before this date.

In the type `BinaryString`, the word `alpha` in round brackets cannot be the name of a bit position and therefore equals `1`.

The `BIT STRING` type with named positions is very often used to indicate the protocol versions supported by a communicating application:

```
Versions ::= BIT STRING { version-1(0), version-2(1) }
```

In the specifications of a multimedia protocol (with PER encoding), we may sometimes come accross types of the following form[15]:

```
BooleanSequence ::= SEQUENCE { b1 BOOLEAN,
                               b2 BOOLEAN,
                               b3 BOOLEAN,
                               b4 BOOLEAN,
                               b5 BOOLEAN,
                               b6 BOOLEAN }
```

whereas, we would rather have written:

```
BooleanVector ::= BIT STRING { b1(0), b2(1), b3(2),
                               b4(3), b5(4), b6(5) }
```

Table 10.2 on the next page shows that the encoding of the `BooleanSequence` type in PER is significantly more compact than for the `BooleanVector` type. But one has only to constrain the size of the latter to realize that the `BooleanSequence` type is not quite a good idea: its PER encoding has exactly the same length (6 bits) as the type `BooleanVector(SIZE (6))`.

We already came up against this problem with the types `ENUMERATED` and `CHOICE` (see Table 10.1 on page 138). It is at least reassuring to know that we can specify in ASN.1 without paying (too much!) attention to the way the data are encoded. We shall see, however, in Chapter 20 that the PER do actually give a really compact encoding only if the types involved are properly subtyped.

Finally, we conclude these examples with the empty string:

```
all-wrong BooleanVector (SIZE (6)) ::= {}
```

In this case, it should be noted that the value will be encoded on 6 bits (in DER and PER notably) as the value `'000000'B`.

[15]This type is assumed to be defined in a module that includes the clause `AUTOMATIC TAGS` in its header.

	BER	**PER**
`BooleanSequence`	20 bytes	6 bits
`BooleanVector`	4 bytes	14 bits
`BooleanVector(SIZE (6))`	4 bytes	6 bits

Table 10.2: Encoding length of a 6-bit string (`'111111'B`)

When it should be transmitted a value specified in an abstract notation other than ASN.1 or encoded according to encoding rules different from those used for the global specification (this value is actually available as a bit or byte string), a subtype constraint such as `CONTAINING/ENCODED BY` (see Section 13.10 on page 283) should be preferred for switching to some other abstract or transfer syntax.

A `BIT STRING` type cannot be compatible with the `OCTET STRING` type introduced in Section 10.7 on page 151.

10.6.2 Reference Manual

Type notation

BitStringType → `BIT STRING`
| `BIT STRING` "{" *NamedBit* "," $\cdots^{+}$ "}"

⟨1⟩ This type has tag no. 3 of class `UNIVERSAL`.
⟨2⟩ This type can be constrained by a single value (production *SingleValue* on page 261), by type inclusion (production *ContainedSubtype* on page 263), by size (production *SizeConstraint* on page 267) and by a constraint on its content (see Section 13.10 on page 283).
⟨3⟩ When a `BIT STRING` type with a named bit list is imported into another module, the identifiers of this list are imported too. These identifiers can only be used to define a value of this type.
⟨4⟩ Two `BIT STRING` types are compatible according to the semantic model of ASN.1 (see Section 9.4 on page 121) even if their associated named bit lists are not identical.
⟨5⟩ When a special encoding is necessary for an abstract value or when a value is specified in a notation different from ASN.1, it is recommended to use a `BIT STRING` or `OCTET STRING` type with a subtype constraint of the form `CONTAINING/ENCODED BY` (see Section 13.10 on page 283).

NamedBit → identifier "(" number ")"
| identifier "(" *DefinedValue* ")"

⟨6⟩ The identifiers of the *NamedBit*s must be distinct, whether they are implicit (number) or referenced (*DefinedValue*).
⟨7⟩ The bit positions of the *NamedBit*s must be distinct.
⟨8⟩ The ordering of the *NamedBit*s is not significant.
⟨9⟩ The maximum value appearing in the list of the *NamedBit*s does not limit the string length.
⟨10⟩ *DefinedValue* must be of type `INTEGER`.
⟨11⟩ *DefinedValue* must be positive.
⟨12⟩ The lexeme *DefinedValue* cannot be interpreted as an identifier of a `BIT STRING` type since an identifier cannot be written in round brackets.

Value notation

BitStringValue → bstring
| hstring
| *IdentifierList*

⟨13⟩ The bit in position `0` is the most left-handed (also called high-order or most significant bit).
⟨14⟩ In the hstring notation, the high-order bit of each hexadecimal digit corresponds to the first bit (left-handed) in the binary string.
⟨15⟩ If the *IdentifierList* is used (and only in this case) the encoding rules may add or remove zeros at the end of the string (trailing bits). Indeed, in this case, only the named positions are significant for the application. Designers should then make sure that the presence or absence of trailing zeros remains meaningless for the application (no semantics should be associated with this). This rule has been changing depending on the versions of the ASN.1 standard; it should therefore be applied with great care.

IdentifierList → "{" identifier "," ···* "}"

⟨16⟩ Each identifier must be the same as one of the identifiers of the corresponding *BitStringType*.
⟨17⟩ The identifiers that are present give the positions of the bits that equal `1`; all the other bits equal `0`.

10.7 The `OCTET STRING` type

10.7.1 User's Guide

In ASN.1, an octet string, like a binary string, can be arbitrarily long. One might have expected the name `BYTE STRING` for this type, but as all memory-words do not have 8 bits on all computers, ISO preferred the French word "*octet*". Indeed, as ASN.1 is meant to be system-independent, the format of a string cannot be restricted to a specific internal representation mode. As a convention, an ASN.1 octet string always comprises (sometimes implicitly) a number of bits that is a multiple of 8.

Generally, the `OCTET STRING` type is used to transmit data that are intrinsically binary (which can be divided into 8-bit packets). An octet string is represented in quotes and followed by the capital letter `B` if it is made of binary numbers (`0` or `1`) or followed by the letter `H` if it is made of hexadecimal digits (from `0` to `9` and from `A` to `F` in upper-case letters only). For the encoding, a sufficient number of `0`s completes the binary string (for example, one trailing `0` is added if the number of hexadecimal components is odd) to reach a number of bits that is a multiple of 8.

```
icon OCTET STRING ::= '001100010011001000110011'B
icon OCTET STRING ::= '313233'H
```

In the binary form, the most left-handed bit is the high-order (or most significant) bit of the first byte. In the hexadecimal notation, the digit is the most significant half-byte of the first byte. Contrary to the `BIT STRING` type, the `OCTET STRING` type does not give the possibility of naming the bit or byte positions by a list in curly brackets.

As the venerable ancestor of ASN.1, the [X.409] standard enabled to specify the octet string in double quotes like a character string. However, the alphabet was not specified and the interpretation of the characters was not straightforward when escape characters were used[16]. This option was not adopted by ASN.1 and such usage is not allowed even though it can be frequently found in Internet RFCs, as [RFC1213] for example.

Concerning the transmission of character strings, ASN.1 provides

[16]The [X.409] standard indicated in its clause 5.4 that "*the interpretation of this second form is context-specific. Every use of it must be accompanied by a detailed specification of the characters that are allowed, their graphical depictions, and their representations as sequences of octets*".

numerous types of character strings described in Chapter 11 and we recommend to use one of those whenever possible: the alphabet is clearly defined and some types use an optimized encoding.

Likewise, the `OCTET STRING` type should be neither used to transmit a value related to another abstract syntax nor encoded according to a transfer syntax different from that of the current specification; neither should it be described in a formalism other than ASN.1. Unfortunately, we very often come across such a use of ASN.1 in telecommunication network signalling for example. In Chapter 14, we will introduce the presentation context switching types, which aim at modeling embedded values.

However, since the constructions mentioned in the previous paragraph are quite often used by designers, the ASN.1 working group now consider defining two new subtype constraints to be associated to the type `OCTET STRING`. This would enable them to specify the type of the embedded value and the encoding rules that are used to produce the octet string (see Section 13.10 on page 283):

```
T ::= OCTET STRING (CONTAINING U ENCODED BY
  {joint-iso-itu-t asn1(1) base-encoding(1)}) -- BER
  -- U is a type defined in the specification
```

More generally, the use of the `OCTET STRING` type should always be the last resort once all the other ASN.1 types have been discarded as inappropriate models of the problem in hand.

The size of the octet strings should be limited using a subtype constraint so that both the encoder and the decoder could allocate the adequate memory space but sometimes also to make the encoding more compact:

```
StringOf5Octets ::= OCTET STRING (SIZE (5))
```

Finally, there is no compatibility between the types `OCTET STRING` and `BIT STRING`

10.7.2 Reference Manual

Type notation

OctetStringType → `OCTET STRING`

⟨1⟩ This type has tag no. 4 of class `UNIVERSAL`.

⟨2⟩ This type can be constrained by a single value (production

SingleValue on page 261), by type inclusion (production *ContainedSubtype* on page 263), by size (production *SizeConstraint* on page 267) and by a constraint on its content (see Section 13.10 on page 283).
⟨3⟩ When a special encoding is necessary for an abstract value or when a value is specified in a notation different from ASN.1, it is recommended to use a `BIT STRING` or `OCTET STRING` type with a subtype constraint of the form `CONTAINING/ENCODED BY` (see Section 13.10 on page 283).

Value notation

OctetStringValue → bstring
| hstring

⟨4⟩ If the binary string does not contain a multiple of 8 bits, a sufficient number of zeros completes the string on the right-hand side (to reach the next multiple of 8).
⟨5⟩ In the bstring notation, the most left-handed bit should be the high-order (or most significant) bit of the first byte.
⟨6⟩ If hstring does not include an even number of hexadecimal digits, a `0` digit completes the string on the right-hand side.
⟨7⟩ In the hstring notation, the most left-handed hexadecimal digit should be the most significant half-byte of the first byte.

10.8 The `OBJECT IDENTIFIER` type

Taxonomy can make your head spin. It does mine whenever my eyes light on an index of the Universal Decimal Classification (UDC). By what succession of miracles has agreement been reached, practically throughout the world, that

668.184.2.099

shall denote the finishing of toilet soap, and

629.1.018-465

horns on refuse vehicles; whereas

621.3.027.23,
621.436:382,
616.24-002.5-084,
796.54, and
913.15

denote respectively: tensions not exceeding 50 volt, the export trade in Diesel motors, the prophylaxy of tuberculosis, camping, and the ancient geography of China and Japan!

Think/Classify, Classifications, by Georges Perec, translated by John Sturrock, in *Species of Spaces and Other Pieces*

10.8.1 User's Guide

Programming languages often have a pointer mechanism to reference variables and structures of the language. The ASN.1 notation offers an even more powerful concept of universal pointers (but without a dereferencing operator as we shall see very shortly): this is the `OBJECT IDENTIFIER` type. In the previous sentence, the term 'universal' refers to the 'physical space' where the reference may point to, as well as to the nature of these objects. Indeed, ASN.1 imposes no constraint on the objects[17] that can be referenced apart from defining such an object as "*a well-defined piece of information, definition, or specification which requires a name in order to identify its use in an instance of communication*".

As for the [ISO9834-1] standard (see below), which gives the general registration procedures for these objects, it defines an object as *"anything in some world, generally the world of telecommunications and information processing or some part thereof, which is identifiable (can be named); and which may be registered"*.

An object identifier can be (the list is not exhaustive):

- an abstract syntax, which must be registered to be used unambiguously on a Presentation connection (layer 6 of the OSI model) [ISO8822, annex B];
- a transfer syntax (or a collection of encoding rules, see footnote 12 on page 15), which must be registered under an unambiguous reference to be able to use abstract syntaxes on a Presentation connection[18] [ISO8823-1, annex B];
- an application entity [ISO7498-3];
- an ASN.1 module (see on page 163);
- a ROSE operation [ISO9072-2] (see on page 80);

[17] We should be cautious about the objects that are discussed here: the object identifiers and the information objects defined in Chapter 15 should not be confused even though an information object can be referenced through an object identifier.

[18] When a connection is established, the Presentation layer is in charge of a negotiation to ensure that the two communicating applications agree on the abstract syntax, the encoding rules and the protocols to be used (see Figure 3.2 on page 22). All these elements are objects identified by object identifiers.

- an attribute of the X.500 directory (see on page 83) to make up a `DistinguishedName`;
- the type of some part of an [X.400] electronic message body (`ExtendedBodyPart`) as in Figure 7.1 on page 82;
- a virtual terminal profile (see on page 81) with its parameters, modes and control characters for example;
- a managed object class, one of its attributes, a notification or even other templates[19] used in the area of network management and the GDMO standard [ISO10165-4] presented in Section 23.3 on page 482.

The registered object should be persistent: an object whose life duration is short should not be registered. In substance, we may say that an object is registered whenever we have to give it a 'name' that should be universally unambiguous and when this needs to be available anywhere in the world.

For example, in the [X.435] standard, which defines the EDI e-mail system (see on page 88), we find the type assignment:

```
EDIBodyPartType ::= OBJECT IDENTIFIER
```

indicating the character set and the EDI standard used in an EDI message body. The [X.435] standard also defines the following body formats:

- EDIFACT: ISO 646|Recommendation T.61|UNDEFINED OCTETS
- ANSIX12: ISO 646|Recommendation T.61|EBCDIC|UNDEFINED OCTETS
- UNTDI: ISO 646|Recommendation T.61|UNDEFINED OCTETS
- PRIVATE: UNDEFINED OCTETS
- UNDEFINED: UNDEFINED OCTETS

Then how should we identify these formats in an unambiguous way? (Remember that the data exchange can be considered on an international level.) How can we include new formats for the specific needs of some company or other without amending the [X.435] standard,

[19]The name given to every template should be unique for a whole network management standard or for a specification restricted to a single company. This name, however, can be used in other specifications although the object identifier associated with this template should remain (universally) unique.

which is a long and tedious procedure?

The naming scheme needed to address these questions should meet the following criterions [PC93]:

- scope: the scheme should enable the naming in a local or international context, and offer this functionality to the greatest number of users (the world-wide context cannot be excluded in the case of open-system interconnection between international companies);
- scale: it has to consider the naming of an unlimited number of objects using potentially complex names;
- persistence: it has to keep the naming as long as necessary and it should permit its removal;
- user-friendliness: it should provide a means of sharing these names on the network but it should also remain intelligible in the specifications.

Several naming schemes could have been proposed but the hierarchical structure of a tree, which satisfies the criterions above, also has the following advantages:

- an object can be found easily when selecting only one branch at each level;
- it reflects the structural properties of the referenced objects but also, if necessary, those of the standardization organizations or administrations;
- this flexibility is not altered as the complexity of the tree increases.

For all these reasons, the ASN.1 standard[20] has defined a naming hierarchy called *registration tree*, whose naming rules (the 'Constitution') come from the [ISO9834-1] standard, which gives the procedures for the OSI registration authorities and defines the registration as "*the assignment of an unambiguous name to an object in a way which makes the assignment available to interested parties*".

Many other formalisms define naming structures with similar properties: SGML, EDIFACT, CDIF, the distinguished names of

[20]In fact, this definition was extracted from the ASN.1 standard and has moved to [ISO9834-1] since late 1996.

X.500 (for which an object identifier counterpart can be defined jointly [ISO9834-1, Annex C]), the addresses of the senders and receivers of an [X.400] e-mail...

The registration tree is based on the same principles as the domains and sub-domains of Internet e-mail addresses. Each registration authority (i.e. an organization, a standard or an automatic procedure that has been agreed on at an international, national or local level) in charge of a tree node respects the following delegated rules:

- it allocates the arcs[21] under its own node and delegates its responsibilities to one or several subordinate authorities[22], which are responsible for these arcs;
- if the arc is a leaf, it is used to name an item of information or a subpart of it (an object) and the registration authority keeps a sgort report for this object (see [ISO9834-1, clause 8], for example);
- it sequentially numbers the arcs starting at `0`;
- it allocates an identifier[23], if needed, to each arc, which is a word beginning with a lower-case letter, including letters, digits and dashes;
- it can sometimes have a more technical role in ensuring that the object can be registered at this level in the tree;
- it maintains a list of the allocated arcs and publishes it regularly;
- if the registration procedures allow it, it deletes and modifies arcs when needed.

All the object identifiers used in this book respect these rules. They all begin with the arc {`iso member-body(2) f(250) type-org(1) ft(16) asn1-book(9)`} that has actually been allocated in the registration subtree of France Télécom.

The top of this registration tree is presented in Figure 10.4 on page 161, with the following comments:

[21] In order to avoid repetitions, we call 'arc' both the nodes and the leaves of the registration tree.

[22] The same authority can obviously be in charge of several nodes.

[23] [ISO9834-1] permits the use of other forms to name the arcs as long as there is no ambiguity. We shall see that this identifier is not compulsory in ASN.1.

- the root has no name but corresponds to the [ISO9834-1] standard;
- the arc `itu-t(0)`[24] is restricted to the ITU-T recommendations (not jointly published with ISO/IEC), to the PTTs and RPOAs (Registered Private Operating Authorities); it is divided into:
  - `recommendation(0)` and the sub-arcs `a(1)` to `z(26)` for each ITU-T recommendation series; then, comes an arc for each recommendation with its number (but no identifier);
  - `question(1)` with, for every four-year period, one branch per commission, whose number is given by the formula 32×period+commission_number, where the period 1984-88 has the number `0` (so that there can be no more than 31 commissions);
  - `administration(2)` reserved for the national administrations of the ITU-T members [X.121];
  - `network-operator(3)`, whose sub-arcs have each a distinct public network code for packet switching (DNIC) [X.121];
  - `identified-organization(4)` for the international research or industry organizations registered at ITU-T;
- the arc `iso(1)` for ISO standards and ISO members; it is managed by ISO and IEC and is divided into:
  - `standard(0)` where we find an arc for every ISO international standard (not jointly published with ITU-T);
  - the arc `registration-authority(1)`, which has never been used and was finally abandoned;
  - `member-body(2)` with an arc for each National Body (every body has a code defined in the [ISO3166-1][25] standard, but no identifier is given to a body by default), which is free to allocate the sub-trees;
  - `identified-organization(3)`, where a range of labels can be allocated to international organizations recognized by ISO, such as the ECMA for example (every one has an international code or ICD, with 4 digits [ISO6523]);

[24]This arc is also called `ccitt(0)` to recall that CCITT used to be an organization independent from ITU-T (see Section 6.2 on page 58).

[25]ftp://ftp.isi.edu/in-notes/iana/assignments/country-codes

- the arc `joint-iso-itu-t(2)`, or `joint-iso-ccitt(2)`, includes around twenty sub-arcs considered as a common standardization area for ISO/IEC and ITU-T; this arc is managed by ANSI according to the [ISO9834-3] standard where we find in particular:
  - the arc `asn1(1)` for the ASN.1 standard, which is managed by the ASN.1 working group at ISO; some sub-arcs identify different encoding rules and it is these object identifiers which are exchanged by the Presentation layers during negotiation (see Figure 3.2 on page 22):
    - `basic-encoding(1)` for the BER,
    - `ber-derived(2)` for the CER and the DER,
    - `packed-encoding(3)` for the different variants of PER (see Chapter 20);
  - the arc `mhs-motif(6)` for the [X.400] e-mail presented in Section 7.2 on page 81;
  - the arc `ms(9)` (management systems) for network management and particularly for the CMIP protocol [ISO9596-1] and the GDMO standard [ISO10165-4] discussed in Section 23.3 on page 482;
  - the arc `registration-procedures(17)` covers the joint activities of ISO and ITU-T on registration procedures.

In the registration tree, the nodes correspond to registration authorities (which can sometimes be represented by a single person) and the leaves are the registered objects. A registered object is denoted uniquely by the names and the numbers of the arcs encountered on the path from the root of the tree to the leaf. This 'name' deduced from the path in the tree is called *object identifier*. It is an ASN.1 value of type `OBJECT IDENTIFIER` represented in curly brackets with no comma.

Here are some examples:

```
internet-id OBJECT IDENTIFIER ::=
  { iso(1) identified-organization(3) dod(6) internet(1) }
francetelecom-id OBJECT IDENTIFIER ::=
  { iso member-body f(250) type-org(1) ft(16) }
ber-id OBJECT IDENTIFIER ::= { 2 1 1 }
```

For the object identifier `internet-id`[26], all the names in curly brackets (also called *identifiers*) are systematically followed by their respective numbers in round brackets. These names make the identifiers more intelligible for the reader of the specification and the numbers are used by the encoding procedures to transmit the value.

In the object identifier `francetelecom-id`, the names `iso` and `member-body` are not followed by their numbers in round brackets. These *standardized identifiers* are assumed to be known by the ASN.1 tools and their number is associated with them automatically by the compiler. These standardized identifiers, for which the number can be omitted, are denoted in yellow on Figure 10.4 on the next page.

Finally the object identifier `ber-id` contains no arc name. Though sufficient to be encoded, this form is strongly unrecommended since it is hardly readable.

[26]In Internet RFCs, the object identifiers are often represented as a character string of type `OCTET STRING` formed by the numbers separated by a dot as in `"1.3.6.1.4.1.1466.115.121.1.5"`. This does obviously not conform to ASN.1 standard, had it only been for the fact that it is impossible to define values of type `OCTET STRING` as character strings!

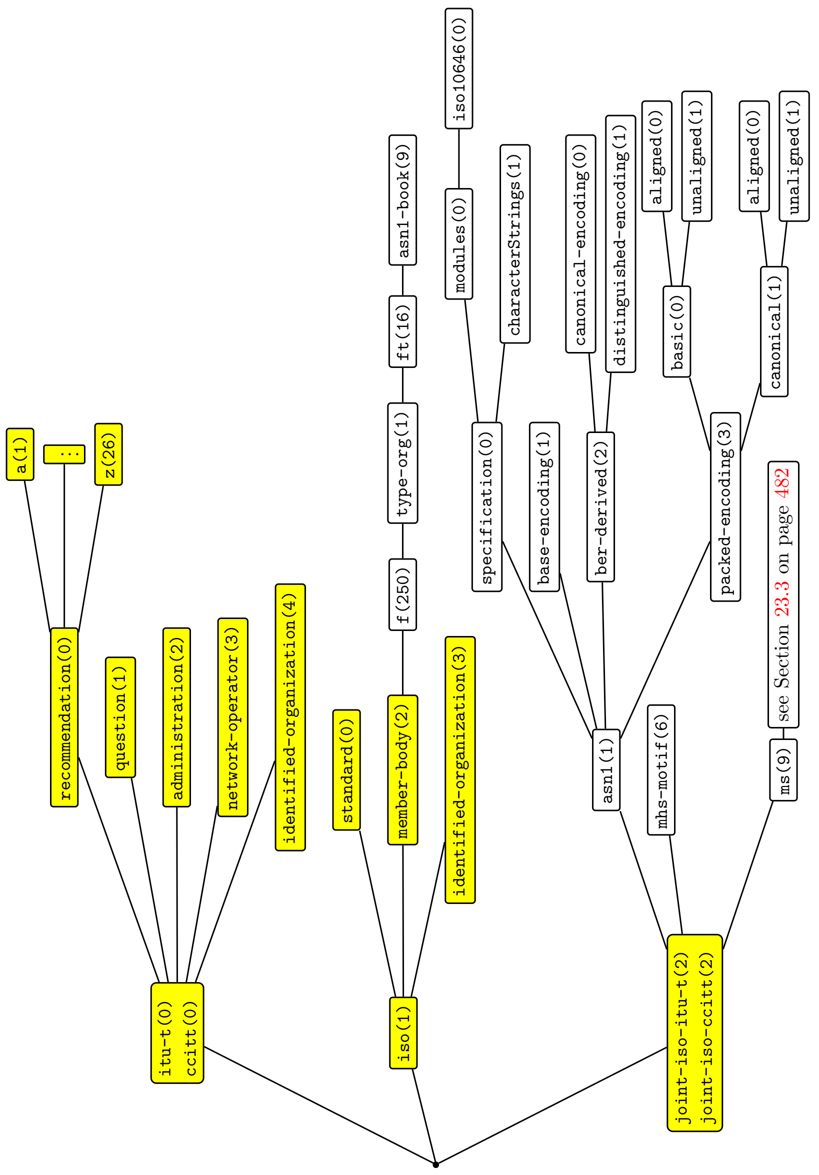


Figure 10.4: Top of the ISO registration tree (identifiers in yellow can be used without their associated number)

When several objects have to be registered under the same node, repeating the absolute path name for every single object identifier can be a tedious work since all of them begin with the same path from the root of the tree. ASN.1 allows the possibility of indicating as the first arc of an object identifier, the name of a value of type `OBJECT IDENTIFIER` whose content is inserted at the beginning of this object identifier.

Hence it can be defined by giving only the relative path from the inserted object identifier (this notion of relative path should not be confused with the `REALTIVE-OID` type defined in the next section; here, the entire path is encoded even if a relative path is used in the abstract notation). As a result, if we go back to the type `EDIBodyPartType` presented on page 155, Appendix A of the [X.435] standard associates the following object identifiers with the various formats of an EDI message body:

```
ID ::= OBJECT IDENTIFIER
id-edims ID ::= { joint-iso-itu-t mhs-motif(6) edims(7) }
id-bp ID ::= { id-edims 11 }
id-bp-edifact-ISO646 ID ::= { id-bp 1 }
id-bp-edifact-T61 ID ::= { id-bp 2 }
id-bp-edifact-octet ID ::= { id-bp 3 }
id-bp-ansiX12-ISO646 ID ::= { id-bp 4 }
id-bp-ansiX12-T61 ID ::= { id-bp 5 }
id-bp-ansiX12-ebcdic ID ::= { id-bp 6 }
```

The object identifier `id-edims` is first inserted at the beginning of the object identifier `id-bp` to give `{ joint-iso-itu-t mhs-motif(6) edims(7) 11 }`, and this object identifier `id-bp` is itself inserted at the beginning of each object identifier of type of formats of EDI message body part (note the breakdown structure!).

Each time an object is registered in the registration tree (and, therefore, each time an object identifier is allocated), the [ISO9834-1] standard recommends[27] to associate a description with it by means of a character string of type `ObjectDescriptor` (see Section 11.15 on page 198). This literal and more user-friendly description is supposed to be universally unique even though it cannot be guaranteed. It is some sort of 'formal' comment that makes the interpretation of the object identifier easier and indicates more clearly what is actually referenced. Thus, the

[27]This advice is actually very rarely taken and information may be added in comments within the ASN.1 module.

object descriptor associated with the object identifier `ber-id` of the BER encoding rules is:

```
ber-descriptor ObjectDescriptor ::=
  "Basic Encoding of a single ASN.1 type"
```

Let us now talk about the more specific use of object identifiers that reference ASN.1 module. When a module is registered in the registration tree, its object identifier is inserted in its header between its name and the keyword `DEFINITIONS`:

```
ModuleName { iso member-body(2) f(250) type-org(1) ft(16)
            asn1-book(9) chapter10(2) module0(0) }
DEFINITIONS ::=
BEGIN
-- ...
END
```

This object identifier can be used for referencing uniquely the module; remember a module name is not necessarily unique (see rule ⟨2⟩ on page 113). Indeed, how could it be imposed on a specifier not to call a module with a name any other specifiers would may have given their modules anywhere else in the world?!

The specifier, of course, registers the module only when it is stable and has no errors of syntax or semantics left (i.e. when it has been checked by an ASN.1 compiler). Besides, a module keeps the same object identifier if it is extended according to the extensibility rules of Section 12.9 on page 244 that define the extension marker "...".

In case definitions should be imported from two different modules with the same name, the object identifier can reference unambiguously each of these modules:

```
Homonym { iso member-body(2) f(250) type-org(1) ft(16)
          asn1-book(9) chapter10(2) homonym1(1) }
DEFINITIONS ::=
BEGIN
T ::= INTEGER
END

Homonym { iso member-body(2) f(250) type-org(1) ft(16)
          asn1-book(9) chapter10(2) homonym2(2) }
DEFINITIONS ::=
BEGIN
T ::= REAL
U ::= BOOLEAN
END
```

```
Module1 DEFINITIONS ::=
BEGIN
IMPORTS T FROM Homonym { iso member-body(2) f(250)
            type-org(1) ft(16) asn1-book(9)
            chapter10(2) homonym1(1) }
        U FROM Homonym { iso member-body(2) f(250)
            type-org(1) ft(16) asn1-book(9)
            chapter10(2) homonym2(2) } ;
V ::= SEQUENCE { integer T,
                 boolean U }
END
```

Moreover, in case (though very unlikely) it is needed to import two definitions with the same name from two modules with the same name, each one of the definitions can be invoked, within the module body that is being specified, by the dotted notation for external references of the form `ModuleName.AssignmentName` (see Section 9.3 on page 117), renaming locally one of the two homonyms in the `IMPORTS` clause:

```
Module2 DEFINITIONS ::=
BEGIN
IMPORTS T FROM Homonym { iso member-body(2) f(250)
            type-org(1) ft(16) asn1-book(9)
            chapter10(2) homonym1(1) }
        T FROM Surname -- renaming -- { iso member-body(2)
            f(250) type-org(1) ft(16) asn1-book(9)
            chapter10(2) homonym2(2) };

W ::= CHOICE { integer Homonym.T,
               real    Surname.T -- local name --}
END
```

Such renaming poses no problem since the object identifier of a module suffices to reference it without ambiguity. Our astute readers may have inferred that such exotic cases rarely occur!

Let us close this section with a few general remarks on object identifiers and registration procedures.

Although object identifiers look like universal pointers, ASN.1 provides no dereferencing operator, i.e. there exists no general means of 'computing' the content of an object if its object identifier is known. Similarly, there exists no world-wide database that we may consult to get the object by simply providing the search engine with an identifier.

Like the Internet domains, the OSI registration tree is distributed: in the strict sense of the word, a node only knows its subordinates. However, an international but distributed directory may be developed relying on the X.500 recommendation (see Section 7.3 on page 83).

As a consequence, a tool (a compiler for instance) cannot check the existence of an object identifier. In practice, the object identifiers used during communication are assumed known by both systems.

It is possible (and some specifications have already abused of this habit even though such practice is inadvisable) to use identifiers (beginning with a lower-case letter) that are different from those already registered for an object identifier, as in:

```
my-per-id OBJECT IDENTIFIER ::= { joint-iso-itu-t
  my-asn1(1) my-packed-encoding(3) }
```

instead of:

```
per-id OBJECT IDENTIFIER ::= { joint-iso-itu-t
  asn1(1) packed-encoding(3) }
```

In such a case, the number associated with each arc should be indicated, including the first two levels if these have been renamed. Such renaming often depends on the business strategy and the public image of companies since some identifiers may have a connotation that they are not necessarily very willing to keep.

It is allowed to register an object several times on different leaves of the tree. It is also possible to propose several identifiers (synonyms) for the same arc even if this is not recommended to avoid confusion in the future.

Finally, in order to reduce the encoding size (for cost reasons but also for hardware restrictions related to chips architecture for instance), a company may be very much interested in having its node as close to the top as possible. For meeting this need but also because there are several categories of registration authorities that cannot be taken into account for the time being in the tree structure, the new `RELATIVE-OID` type has been introduced in 1999. It will be presented in the next section.

10.8.2 Reference Manual

Type notation

ObjectIdentifierType → `OBJECT IDENTIFIER`

⟨1⟩ This type has tag no. 6 of class `UNIVERSAL`.
⟨2⟩ This type can be constrained by a single value (production *SingleValue* on page 261) and by type inclusion (production *ContainedSubtype* on page 263), even though it is useless in practice. It is recommended to use the new `RELATIVE-OID` type instead (see Section 10.9 on the next page).

Value notation

ObjectIdentifierValue →
  "{" *ObjIdComponents* $\cdots^{+}$ "}"
| "{" *DefinedValue* *ObjIdComponents* $\cdots^{+}$ "}"

⟨3⟩ In the second alternative, the reference *DefinedValue* should only denote a value of type `OBJECT IDENTIFIER` or `INTEGER` (see also rules ⟨5⟩ and ⟨8⟩ below). When *DefinedValue* denotes an `OBJECT IDENTIFIER` value, the list of its arcs prefixes the arcs explicitly present in the *ObjectIdentifierValue*.
⟨4⟩ *ObjectIdentifierValue* must contain, maybe once the *DefinedValue* has been developed, at least two *ObjIdComponents*. This restriction is induced by the BER encoding rules that encode together the first two arcs of an object identifier (see Section 18.2.8 on page 404).

ObjIdComponents → *NameForm*
  | *NumberForm*
  | *NameAndNumberForm*
  | *DefinedValue*

⟨5⟩ *DefinedValue* denotes here a value of type `RELATIVE-OID` (see also rules ⟨3⟩ and ⟨8⟩ on the next page). In this case, the *ObjIdComponents*, which necessarily precede it, define the reference node from which the relative object identifier should be concatenated (this reference node cannot be the root of the registration tree, nor a node immediately under this root, see rule ⟨4⟩ on the current page); the following *ObjIdComponents* (if present) denote the arcs below the last node of this relative object identifier.

NameForm → identifier

⟨6⟩ An identifier should be followed by a bracket (production *NameAndNumberForm*) except if it is one of the identifiers in yellow in Figure 10.4 on page 161. The integer associated with every one of these identifiers in yellow is assumed to be known by ASN.1 tools. This

integer is set once for all in the [ISO9834-1] standard and cannot be changed.
⟨7⟩ As the identifier does not constrain the encoding, it can be modified for 'political' reasons that may concern the organization (even if this is deprecated). In this case the production *NameAndNumberForm* must be used to indicate the associated number.

$$NumberForm \rightarrow \text{number} \mid \underline{DefinedValue}$$

⟨8⟩ *DefinedValue* references a value of type `INTEGER` (see rules ⟨3⟩ and ⟨5⟩ on the preceding page).
⟨9⟩ The integer referenced by *DefinedValue* must be positive.
⟨10⟩ If *DefinedValue* is both the reference of a value defined in the current module and one of the identifiers in yellow on Figure 10.4 on page 161, the latter prevails.

$$NameAndNumberForm \rightarrow \text{identifier} \text{ “(” } NumberForm \text{ “)”}$$

10.9 The `RELATIVE-OID` type

10.9.1 User's Guide

In an instance of communication, many transmitted object identifiers often denote objects registered in the same sub-tree of the registration tree. Otherwise said, all these identifiers *relate* to a common *reference node*.

In this case, the volume of data to be transmitted can be cut down by factorizing the beginning of the path from the root of the tree up to this reference node. For hardware restrictions for example, it is necessary to reduce the encoding size when transferring data to a chip card or from satellites. This new type will therefore be appreciated in industries using narrow bandwidth transmission and real-time systems (Universal Postal Union, intelligent transportation systems, radio-frequency identification RFID, etc).

Issued from the Beijing meeting in September 1998, the `RELATIVE-OID` type was finalized in Geneva in June 1999. This swiftness of action in designing the Amendment [ISO8824-1Amd1] shows how quickly the ASN.1 workgroup can respond to specifiers' expectations regarding the features offered by the standard.

Thus, in the field of radio-frequency identification (see on page 92), electronic tags can provide several information containers. A whole class of containers, for example, can be referenced by a single object identifier while every one of its individual containers is identified by a relative object identifier, thereby reducing the data size when several containers of this class have to be transmitted.

There exists no way of associating formally (i.e. denoting) the reference node's object identifier to a `RELATIVE-OID` type in ASN.1 specifications. Indeed, in practice this object identifier is always known in one way or another by the two communicating applications. In the rest of this section, we shall give a few tips for using this type to illustrate our point.

The type:

```
RELATIVE-OID -- { iso member-body(2) f(250) type-org(1)
             --   ft(16) asn1-book(9) }
```

allows only (relative) object identifiers registered in the sub-tree whose object identifier is indicated in comments (all the object identifiers assigned to this book are therefore appropriate values for this type). Only the final part is transmitted. As shown in the example above, comments are inserted in the ASN.1 module to declare the reference node. This alternative should be retained for specifications related to a particular application domain or if these data are static, that is to say if they do not change during the communication.

In other cases, a basic standard provides a generic protocol using a parameterized type (in the same way as ROSE, where the parameter is an information object set, see Chapter 17). Then a specific standard imports this parameterized type so that all the object identifiers of this standard have a common root that is statically known and thus needs not be encoded.

A most complex case arises when the reference node's object identifier has to be communicated at 'run-time'. The comment associated with the `RELATIVE-OID` type would then indicate the name of some other component in charge of the transmission of this object identifier as in:

```
SEQUENCE {
  reference-node OBJECT IDENTIFIER DEFAULT { iso
          member-body(2) f(250) type-org(1)
          ft(16) asn1-book(9) },
  relative-oids    SEQUENCE OF RELATIVE-OID
                  -- relative to reference-node -- }
```

The reference node is only present dynamically during communication; it is shared by all the relative object identifiers provided to the component `relative-oids`.

One may also use a user-defined constraint introduced by the keywords `CONSTRAINED BY` (see Section 13.13 on page 294).

Finally, an application can optimize the relative object identifier transmission when the root is known beforehand while keeping the possibility of using absolute object identifiers. To do so, we use the type:

```
CHOICE { absolute-oid  OBJECT IDENTIFIER,
         relative-oids RELATIVE-OID }
```

The root constraint (introduced by the keyword `ROOT`) that was proposed at the same time as the `RELATIVE-OID` type and whose purpose was to restrict the set of values of the types `OBJECT IDENTIFIER` and `RELATIVE-OID` has not been standardized in the end.

10.9.2 Reference Manual

Type notation

RelativeOIDType → `RELATIVE-OID`

⟨1⟩ The `RELATIVE-OID` type has tag no. 13 of class `UNIVERSAL`. It has been introduced in 1999 by an amendment on ASN.1:1997 [ISO8824-1Amd1].
⟨2⟩ This type can be constrained by a single value (production *SingleValue* on page 261) and by type inclusion (production *ContainedSubtype* on page 263). The reference node of all the values and all the types appearing in subtype constraints (and therefore in value sets of `RELATIVE-OID` type, see on page 333) must be the same as the reference node of the governing type.
⟨3⟩ The reference node of all the relative object identifiers (i.e. the node from which the relative object identifiers start) must be denoted in a comment within the specification (see rule ⟨5⟩ on page 101), in the documentation associated with the protocol or with an object identifier transmitted in the same communication instance. This reference node cannot be the root of the registration tree nor a node immediately beneath this root (see rule ⟨4⟩ on page 166).

Value notation

RelativeOIDValue → "{" *RelativeOIDComponents* $\cdots^{+}$ "}"

$$
\begin{array}{rcl}
\textit{RelativeOIDComponents} & \rightarrow & \textit{NumberForm} \\
& \mid & \textit{NameAndNumberForm} \\
& \mid & \textit{DefinedValue}
\end{array}
$$

⟨4⟩ *DefinedValue* references a value of type `RELATIVE-OID` (see also rule ⟨5⟩ on this page). In this case, the *RelativeOIDComponents* that may precede it define the (relative) reference node from which the relative object should be concatenated (this node cannot be the root of the registration tree nor a node immediately beneath this root, see rule ⟨4⟩ on page 166); the *RelativeOIDComponents* that may follow it denote the arcs under the last node of this referenced relative object identifier.

$$
\begin{array}{rcl}
\textit{NumberForm} & \rightarrow & \mathsf{number} \\
& \mid & \underline{\textit{DefinedValue}}
\end{array}
$$

⟨5⟩ *DefinedValue* is here a reference to a value of type `INTEGER` (see also rule ⟨4⟩ on this page).

⟨6⟩ The integer referenced by *DefinedValue* cannot be negative.

$$\textit{NameAndNumberForm} \rightarrow \mathsf{identifier}\ \text{“}\texttt{(}\text{”}\ \textit{NumberForm}\ \text{“}\texttt{)}\text{”}$$

⟨7⟩ The identifier does not influence the encoding, thus it may not necessarily be the one associated with *NumberForm* when registering the object. Such a practice, however, is not recommended since it does not go along with a better readility of the specifications.

Chapter 11

Character string types

Contents

But the mere fact that there is an order no doubt means that, sooner or later and more or less, each element in the series becomes the insidious bearer of a qualitative coefficient [...].

The qualitative alphabetical code is not very well stocked. In fact, it has hardly more than three elements:
A = excellent [“class A” cigarettes];
B = less good;
Z = hopeless (a Z-movie).

Georges Perec, *Think/Classify, The Alphabet*, translated by John Sturrock, in *Species of Spaces and Other Pieces.*

The diversity in character strings offered by ASN.1 provides enough material for a whole chapter, and even more so since the ASN.1 standard often calls for other ISO standards to define them.

Once exposed general principles on character strings, we present two types specific to ASN.1, then six types based on the *ISO International Register of Coded Character Sets with Escape Sequences* [ISOReg][1] and finally three types introduced in 1994 and adapted from the [ISO10646-1] standard. We conclude with three special types, two of which are used for handling dates and times in particular.

11.1 General comments

When using character strings, the ASN.1 specifier encounters problems which are inherently due to character string handling (that is, those programmers are naturally familiar with, but also difficulties more specifically related to data transmission).

[1]All the entries of the [ISOReg] register are available at http://www.itscj.ipsj.or.jp/ISO-IR/. General information about character strings (particularly the ASCII, [ISO646], [ISO8859] and [ISO10646-1] alphabets) can be found at http://www.terena.nl/projects/multiling/ml-mua/mlmua-docs.html.

The definition of a character string (an abstract value) in an ASN.1 module is written in double quotes. If the string itself includes quotes, these should be doubled:

```
string IA5String ::=
   "string including ""double quotes"""
   -- stands for <<string including "double quotes">>
```

A string may spread over several lines (i.e. contain newline characters); in this case, spaces and tabulations at the beginning and the end of a line as well as these newline characters are ignored. In the string below, the spaces emphasized by the symbol "␣" (and the newlines) are not considered as being part of the string:

```
pi-in-base-26 PrintableString ::= "d,drsqlolyrtrodnlhn␣
␣␣␣qtgkudqgtuirxneqbckbszivqqvgdmelmsciekhvdutcxtjpsb␣␣
␣␣␣whufomqjaosygpoupymlifsfiizrodplbjfgsjhn"
```

This remark applies only on character strings explicitly written in an ASN.1 module (as abstract values), but do not on those provided by an application to an encoder.

The characters in strings are to be interpreted accordingly to the ASN.1 type of the string[2]. Control or escape characters cannot be inserted in a character string abstract value (but they can be encoded all the same); we shall see that ASN.1 provides, for four different character string types, a particular syntax for denoting these special characters (see production *CharsDefn* on page 196). The management of escape characters is not ensured by the encoders and the decoders but is left to the applications.

The diversity of character string types prevents the encoder or decoder from checking the conformity of every one of them to their alphabet. To avoid interworking problems, these should be subtyped whenever it is possible, using a size constraint with the keyword `SIZE` (see Section 13.5 on page 266) and an alphabet constraint with the keyword `FROM` (see Section 13.6 on page 268).

Furthermore, for some types (see Table 11.1 on page 175), the encoding size of a character is not constant and depends on the character involved (we say that such types are *not known-multiplier character*

[2]But some types include characters with the same *glyph* (graphical symbol) such as the capital letters alpha and "`A`" (i.e. they look the same 'on paper'). In this case, the type `UniversalString` (see Section 11.10 on page 183) can provide unambiguous references like `latinCapitalLetterA` and `greekCapitalLetterAlpha` to distinguish between the two characters.

string types). In this case, it is impossible to decode the nth character without decoding and interpreting the first $(n-1)$th.

Although considered as ASN.1 keywords since 1994, the names of the character string types are not in capital letters because ASN.1 historically defines them using the `OCTET STRING` type[3] in definitions like:

```
IA5String ::= [UNIVERSAL 22] IMPLICIT OCTET STRING
```

Since each type has its own tag in the `UNIVERSAL` class (see Table 11.1 on the next page), it can be provided with its specific encoding rules.

The types that are generally recommended are `IA5String` and `UTF8String`.

Describing precisely and completely which characters make up a given alphabet can be a very fastidious and hopeless work and this chapter is not meant to be (and ought not be so for the reader's sake) exhaustive.

Contrary to the partitioning adopted in the other chapters of this second part of the book, the Reference Manual of the various character string type and value notations is set back to Section 11.13 on page 192.

11.2 The `NumericString` type

As defined in the [X.409] standard, this type "*models data entered from such devices as telephone handsets*". The corresponding alphabet consists of the space character and the digits from "`0`" to "`9`"[4].

The `NumericString`[5] type is a known-multiplier character string type (every character is encoded on the same length). It can be used to model any kind of identification number (ID, credit card number, etc):

```
IDnumber ::= NumericString
```

Note that spaces are valid in a `NumericString` value.

[3]These definitions are useless in practice because, as mentioned in Section 10.7 on page 151, it is not allowed to write an `OCTET STRING` in double quotes.

[4]The well-known keys "`*`" and "`#`" had been forgotten because the telephones were buttonless at that time!

[5]It has the object descriptor (see Section 11.15 on page 198) `"NumericString character abstract syntax"` and is registered with the object identifier `{joint-iso-itu-t asn1(1) specification(0) characterStrings(1) numericString(0)}`.

Type name	***tag***[a]	**Alphabet**	**ESC**[b]	**mult.**[c]
`NumericString`	18	"0" to "9", space		√
`PrintableString`	19	"A" to "Z", "a" to "z", "0" to "9", space, "'", "(", ")", "+", ",", "-", ".", "/", ":", "=", "?"		√
`VisibleString` `ISO646String`	26	[ISOReg] entry no. 6; space		√
`IA5String`	22	[ISOReg] entry no. 1 & 6; space, delete (see Table 11.2 on page 178)		√
`TeletexString` `T61String`	20	[ISOReg] entry no. 6, 87, 102, 103, 106, 107, 126, 144, 150, 153, 156, 164, 165, 168; space, delete (in ASN.1:1990, only the [ISOReg] entries no. 87, 102, 103, 106 & 107 were allowed)	√	
`VideotexString`	21	[ISOReg] entry no. 1, 13, 72, 73, 87, 89, 102, 108, 126, 128, 129, 144, 150, 153, 164, 165, 168; space, delete	√	
`GraphicString`	25	all the graphical sets (called 'G') of [ISOReg]; space	√	
`GeneralString`	27	all the graphical sets (called 'G') and all the control characters (called 'C') of [ISOReg]; space, delete	√	
`UniversalString`	28	[ISO10646-1]		√
`BMPString`	30	the basic multilingual plane [ISO10646-1] (65,536 cells)		√
`UTF8String`	12	[ISO10646-1]		

[a]`UNIVERSAL` class tag number

[b]Escape characters allowed

[c]Known-multiplier character string type (every character of the string is encoded on the same number of bytes)

Table 11.1: The alphabets of the character string types (from the particular to the general)

11.3 The `PrintableString` type

The standard ASN.1:1984 [X.409] used to present it as the type of "*data entered from devices with a limited character repertoire (for example, Telex terminals)*". The corresponding alphabet consists of spaces, upper-case and lower-case letters, digits and the symbols "`'`", "`(`", "`)`", "`+`", "`,`", "`-`", "`.`", "`/`", "`:`", "`=`" and "`?`". For some spoken languages, it can be used to model surnames and first names (provided they do not include accents, among other things), but it is not suitable for an e-mail address[6], for example.

Since 1994, an order relationship has been defined on the `PrintableString` alphabet (the characters are listed in increasing order on Table 11.1 on the preceding page). This is meant to limit the alphabet to a certain interval as we shall see in Chapter 13 on subtype constraints:

```
CapitalLettersAndSpaces ::=
  PrintableString (FROM ("A".."Z"|" "))
```

`PrintableString`[7] is a known-multiplier character string type.

11.4 The `VisibleString` and `ISO646String` types

These two types are similar (but it is recommended to use the name `VisibleString` because the [ISO646] standard is not referenced in the type definition any more); as a consequence, they have the same tag no. `26` of class `UNIVERSAL`. Their alphabet is the international register[8] no. 6[9] [ISOReg], plus the space. As self-explicitly indicated by its label,

[6]However, [RFC2156] provides a translation mechanism where "`@`" is represented by "`(a)`", "`!`" by "`(b)`", "`_`" by "`(u)`", "`(`" by "`(l)`"... in order to draw a correspondence between the Internet ASCII encoding and the `PrintableString` type found in some items of the X.400 Directory.

[7]It has the object descriptor `"PrintableString character abstract syntax"` (see Section 11.15 on page 198) and is registered with the object identifier {`joint-iso-itu-t asn1(1) specification(0) characterStrings(1) printableString(1)`}.

[8]Every entry of the [ISOReg] register stands for a complete character set, with the same structure as the ASCII table on page 178, i.e. 128 octets in 8 columns and 16 rows. The control characters are in the first two columns (in yellow) and the DEL character is always in the last cell at the bottom on the left-hand side. Whatever the register, the escape character ESC is always in the same place.

[9]Until 1992, the ASN.1 standard has referenced the ISO 646:1983 standard (which is actually the international register no. 2). In 1990, the [ISO646] standard was revised and it now corresponds to the international register no. 6 (the character "`$`" was the

strings of type `VisibleString` include no escape characters, no newlines nor any combination such as those for obtaining the accents with the backspace, for example. Finally, these are known-multiplier character string types.

11.5 The `IA5String` type

The 'International Alphabet number 5' (or IA5) is based on 7-bit characters and was jointly published by ISO and ITU-T (recommendation T.50) in 1963. It has become the basic character set of most of the communicating systems. It is generally equivalent to the ASCII alphabet (international standard de facto), but national versions, which can take into account accents or characters specific to some spoken languages may be proposed by national standardization organizations. As seen in Table 11.2 on the next page, the alphabet `IA5String` consists of 128 characters altogether divided into 3 groups:

- a set (called C0) of 32 control characters (positions[10] from 0/0 to 1/15 in yellow);
- a set (called G) of 94 graphical (i.e. visible) characters in which 10 of them are kept for national use or are specific to the communicating application;
- two special characters: space in position 2/0 (32 in decimal) and delete in position 7/15.

The control character set C0 is an heterogeneous collection of functions among which we find the formats (CR, LF, BS, HT, VT, SP, FF), the extension functions (SO, SI, ESC) that enable extending the character set according to the [ISO2022] standard when the default set G is not sufficient[11], the information separators, the communication functions (ACK...). The set C0 is very often used in computing but new transmission techniques or protocols made redundant or even incompatible many

only addition!). If it is the former definition of `VisibleString` and `ISO646String` that is meant to be referenced in a specification, the `CHARACTER STRING` type described in Section 14.3 on page 306 can be used.

[10]The positions of the characters are given in the form 'column/row' with respect to the 8 (columns) by 16 (rows) table on page 178.

[11]It is the case for accentuated letters which are characters made of a combination of a letter, the backspace character (BS) and the appropriate accent.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	NUL[a]	DLE	SP	0	[b]	P	[b]	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#/£ [c]	3	C	S	c	s
4	EOT	DC4	¤/$ [c]	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
10	LF	SUB	*	:	J	Z	j	z
11	VT	ESC[d]	+	;	K	[b]	k	[b]
12	FF	IS4	,	<	L	[b]	l	[b]
13	CR	IS3	-	=	M	[b]	m	[b]
14	SO	IS2	.	>	N	[b]	n	[b]
15	SI	IS1	/	?	O	_	o	DEL [d]

[a]The columns in yellow correspond to the control character set C0.

[b]This position is free for national or application-specific use.

[c]One of these two characters must be chosen in any specific use of this alphabet.

[d]The characters ESC and DEL are always found at the same position in all the character tables extracted from the [ISOReg] register.

Table 11.2: The `IA5String` alphabet

of these. A few of codes also became useless because some applications had used them abusively in the sense that their interpretation relied in fact on a tacit agreement between the sending and receiving application.

The types `NumericString`, `PrintableString` and `VisibleString` previously introduced are obviously subtypes of the `IA5String` type.

Since ASN.1:1994, the character set C0 has been defined in the standardized module called `ASN1-CHARACTER-MODULE`[12] by values of type `IA5String` like:

```
cr IA5String ::= {0,13}
del IA5String ::= {7,15}
```

The 2-tuple in curly brackets is an ASN.1 notation specific to the `IA5String` type, which allows referencing non-graphical characters by their positions in the Table 11.2 on the preceding page. This notation (described in Section 11.13 on page 192) can, of course, be used by the ASN.1 specifier. If non graphical characters are to be inserted in a character string of type `IA5String`, the following notation can be used once the adequate character references have been imported:

```
ExampleIA5String DEFINITIONS ::=
BEGIN
IMPORTS cr FROM ASN1-CHARACTER-MODULE {joint-iso-itu-t
    asn1(1) specification(0) modules(0) iso10646(0)} ;

two-lines IA5String ::= { "First line", cr,
                          "Second line" }
END
```

The value `two-lines` contains the newline character (CR).

11.6 The `TeletexString` and `T61String` types

These two types are similar (even though it is recommended to use the name `TeletexString`); as a consequence, they have the same tag no. 20 of class `UNIVERSAL`. The Teletex was designed as a 'super-telex' service for inter-connecting word-processing machines according to a page-based transmission mode with an alphabet of 308 characters. It was divided into five blocks [ISO6937]:

- a control character primary set and an additional set (the columns 0, 1, 8 and 9 of a 15-row table similar to Table 11.2 on the preceding page);

[12] http://asn1.elibel.tm.fr/en/standards/ASN1-CHARACTER-MODULE.asn

- a graphical character primary set and an additional set (the columns 10 to 15);
- the space and the delete character.

Using characters encoded on 8 bits (or 16 bits for composed characters), the `TeletexString` type doubles the possibilities offered by the `IA5String` type (even if it does not quite use the same characters); it makes it possible to use text transmission in different languages more easily (the column 12 includes diacritical characters that accentuate letters when they prefix graphical characters of the primary set, i.e. the Roman alphabet). These can frequently be found in specifications based on the X.400 (electronic mail) and X.500 (directory) standard series. Despite some undeniable advantages, it is seldom used today.

11.7 The `VideotexString` type

The Videotex system enables the user to visualize on a television screen or any equivalent terminal such as the French Minitel (see Figure 11.1 on the next page), numerical text or graphical information transmitted on the telephone network. The user employs a return channel to inter-act with the computer on which the information is stored.

The [T.101] standard on interworking of Videotex services at an international level emphasizes that: "*a `VideotexString` within ASN.1 consists of a string of characters selected from the data syntaxes I, II and III of* [Recommendation T.101] *or from the annexes defining common Videotex components (audio, photographic). These are defined in registration 131, 145 and 108 under [ISO2375]. The common component of audio and photographic are registered under ISO 9281, Part 2*".

The data syntax I is the the data syntax for the Japanese CAPTAIN videotext system ('Character And Pattern Telephone Access Information Network system'). Created in 1978, it is adapted for Kanji ideograms and the Katakana alphabet but it also includes the IA5 alphabet, mosaics, dynamically re-definable characters (or 'soft' characters) and codes for controls, size, flickering...

The data syntax II is the old data syntax of CEPT ('Conference of European Postal and Telecommunications Administrations'). Using five recommendations published in 1998, it includes the Chinese ideograms,

© France Télécom R&D/Michel Le Gal

Figure 11.1: An example of a Videotex terminal: the French Minitel ®

many symbols, mosaic graphical characters such as ⊟, graphical characters for line drawing (which describes a drawing using a series of instructions for an arc, a finite line, a contour or a filled surface) as well as formatting codes of colour, size or scrolling...

The data syntax III is the data syntax of the American NAPLPS videotext system ('North American Presentation Level Protocol Syntax'). Created in 1982, it includes the Latin alphabet, many symbols and graphical characters.

The characters of this type are encoded on 7 or 8 bits; it is not a known-multiplier character string type. The `VideotexString` character strings may include escape characters.

This type is no longer used.

11.8 The `GraphicString` type

A character string of type `GraphicString` can include spaces and any of the graphical (i.e. visible) character sets (called "`G`") registered in the 'International Register of Coded Character Sets to be used with Escape Sequences' [ISOReg]. This international register collects character sets of common usage established in accordance to the [ISO2022] standard ('Character code structure and extension techniques'), and registered[13] according to the [ISO2375] standard ('Procedure for registration of escape sequences'). To switch from one to the other, an escape sequence has to be used (see the ESC character of the IA5 alphabet in Table 11.2 on page 178).

This escape mechanism and the generality of the character set to be handled can obviously be a problem to assess the validity of an encoding and one could say that no ASN.1 tool fully takes into account the `GraphicString` type even though it was very much used in the first specifications of the OSI world. Moreover, the interpretation of a `GraphicString` by a receiving application proves more computationally expensive since the characters are not encoded on a constant number of bytes.

We shall see in the next section that `UniversalString` or `BMPString` types solve all these problems without restricting the `GraphicString` type's generality. Using `GraphicString` is therefore not recommended any more.

11.9 The `GeneralString` type

The `GeneralString` type is based on all the character sets of the `GraphicString` type described above and includes all the control character sets (called "C") of the [ISOReg] standard.

Like the `GraphicString` type, `GeneralString` is too general to be implemented. It was, for example, ill-used in the first version of the Z39.50 protocol (see on page 87):

```
    InternationalString ::= GeneralString
```

because no agreement could be found on an alphabet. Today, its use is not recommended.

[13]Potentially, anyone can define a new set of characters, register it and use an escape sequence for selecting it.

11.10 The `UniversalString` type

11.10.1 User's Guide

At the end of the 1980s, the subcommittee 2 (SC 2) of JTC 1 (see Figure 6.2 on page 56), in charge of all the character set standards, initiate a very ambitious program in order to design a single structure that would take into account all the alphabets of all the languages on Earth! Sometimes questioned, this work ended in late 1992 by the publication of the [ISO10646-1][14] standard, also known as *Unicode* [Uni96][15] (presented in [Cha97a], for example).

The [ISO10646-1] standard, called 'Universal multiple-octet coded Character Set' (UCS), potentially offers 2^{31} cells (each of which contains a single character) stratified into 128 groups of 256 planes of 256 rows of 256 cells (i.e. an encoding of four bytes at most for each cell). At the moment, only the first plane (38,885 cells), called Basic Multilingual Plane or BMP, is allocated. Its rows contain the following alphabets:

- the first 127 characters are those of ASCII, i.e. the international reference version of the IA5 alphabet (see Table 11.2 on page 178);
- the second half of the first row re-use the Latin 1 characters; this alphabet is a set of 8-bit characters defined as a superset of ASCII to take into account most of the national European character set (the accentuated characters, in particular);
- the accentuated characters of Eastern Europe;
- the international phonetic alphabet;
- the Greek alphabet (with the two kinds of accentuated characters);
- the Cyrillic, Georgian and Armenian alphabets;

[14]The number 10646 was given to recall the ISO 646 standard which is just beneath the surface of many character sets, and more particularly ASCII and the International Register [ISOReg].

[15]Since 1993, the standards [ISO10646-1] and [Uni96] have included exactly the same character sets and the modifications of these are carried out simultaneously on both texts. Nevertheless, [Uni96] defines extra properties on characters and additive specificities from communicating applications that can convey a great interest for programmers. Besides, [Uni96] does not offer the UCS-4 encoding on four bytes but only the UCS-2 encoding on two bytes. A short introduction of these two standards is given at http://www.nada.kth.se/i18n/ucs/unicode-iso10646-oview.html.

- the Hebrew alphabet;
- the four types of Arab characters;
- the languages used on the Indian sub-continent;
- the Thai and Lao;
- the Chinese, Corean and Japanese ideograms;
- logical and arithmetic operators;
- characters to draw lines and boxes;
- geometrical forms and dingbats;
- characters dedicated to check optical recognition;
- circled characters.

The [ISO10646-1] standard should become the basic encoding of 16- or 32-bit computers[16]. It will make it possible to use a software from one country to another because of the following properties:

- the encoding scheme is uniform for all characters (whether they are alphabetical characters, symbols or ideograms) and unambiguous;
- the length of encoding is fixed (the UCS-4 canonical form encodes each character on four bytes, but we shall see in the next section the less expensive UCS-2 form for the `BMPString`);
- a string is easily decoded since the escape character and the control characters are not used[17];
- efficient sort and search routines can be applied on the character strings.

[16]It has been adopted by Microsoft Windows® and by Java® of Sun Microsystems Inc., for example.

[17]Note, however, that these characters are allowed since they appear in the [ISO10646-1] standard. We will see in Chapter 18 that ASN.1 encoding standards restrict their use.

With every character, the standard [ISO10646-1] associates a numerical value represented as a quadruple ⟨group, plane, row, cell⟩ and sometimes (for little less than 6,000 characters) a name[18] such as Latin Capital Letter A or Greek Capital Letter Alpha. It also gives a name to the 84 most commonly used character subsets like Latin1[19], Latin2, Latin3, Katakana... An appendix of the [ISO10646-1] standard provides an algorithm to assign an object identifier (see Section 10.8 on page 153) to every possible combination of these defined subsets.

Finally, the [ISO10646-1] standard describes three implementation levels:

- level 1: combining characters[20] are not allowed; all the characters are individually visible;
- level 2: only the combining characters of [ISO10646-1, annex B] are allowed;
- level 3: there is no restriction on the combining characters.

In ASN.1, the [ISO10646-1] standard has been represented by the `UniversalString` type since 1994.

ASN.1 allows referencing a particular character by the corresponding quadruple ⟨group, plane, row, cell⟩ as defined above, using curly brackets:

```
latinCapitalLetterA UniversalString ::= {0,0,0,65}
greekCapitalLetterSigma UniversalString ::= {0,0,3,145}
```

Besides, the ASN.1 standard [ISO8824-1, clause 37] gives an algorithm to derive automatically an ASN.1 value name of `UniversalString` type[21] for each cell of the [ISO10646-1] standard. All these names are collected in a standardized module called `ASN1-CHARACTER-MODULE`[22] so

[18]These can be obtained at ftp://ftp.unicode.org/Public/UNIDATA/UnicodeData-Latest.txt or http://www.unicode.org.

[19]The subset Latin1, for example, concerns western Europe area and gathers the following languages: German, English, Danish, Spanish, Faroe Islands', Finnish, French, Dutch, Irish, Icelandic, Italian, Norwegian, Portuguese, and Swedish [ISO8859].

[20]We call 'combining character', a character like the backspace (BS, see Table 11.2 on page 178), which can represent an accentuated letter using two other characters (the letter and the accent).

[21]In fact, these values belong to the `BMPString` type, but this is not a problem for writing abstract values since these two types are equivalent as far as the abstract syntax is concerned (see Section 11.14 on page 197).

[22]http://asn1.elibel.tm.fr/en/standards/ASN1-CHARACTER-MODULE.asn

that the specifier only needs to import the corresponding character reference from this module:

```
MyModule DEFINITIONS ::=
BEGIN
IMPORTS latinCapitalLetterA, greekCapitalLetterAlpha
  FROM ASN1-CHARACTER-MODULE {joint-iso-itu-t
    asn1(1) specification(0) modules(0) iso10646(0)};

my-string UniversalString ::= {
  "This is a capital A: ", latinCapitalLetterA,
  ", and a capital alpha: ", greekCapitalLetterAlpha,
  "; try and spot the difference!"}
END
```

The value `my-string` introduces a specific syntax (already used on page 179 for the `IA5String` type) for mixing ('concatenating') within curly brackets, character strings (in double quotes) and character references (imported from the module `ASN1-CHARACTER-MODULE`). This notation enables the specifier to make a distinction between different characters represented by the same glyph (like the capital A of the Latin alphabet and the capital alpha of the Greek alphabet).

Finally, the character subsets that are the most commonly used and that are mentioned in the [ISO10646-1] standard are also referenced in the module `ASN1-CHARACTER-MODULE`, where they are defined as subtypes of the `UniversalString` (actually `BMPString`) type. They can be invoked in subtype constraints by alphabet restriction or by type inclusion for an easy restriction of the `UniversalString` type it will be illustrated on page 262. Indeed, the [ISO10646-1] standard (and this is also mentioned in the ASN.1 standard) deprecates any unrestricted use of this type and recommends that the subset implemented should be clearly stated together with its implementation level[23]. One may limit the `UniversalString` type to the level 1 implementation of the `Latin1` alphabet as in:

```
Latin1Level1 ::= UniversalString
                   (FROM (Latin1 INTERSECTION Level1))
```

[23]The characteristics of a standard (i.e. the way an implementation comply with a standard) are indicated in a protocol implementation conformance statement (PICS, see footnote 4 on page 379).

The subtyping also relieves the specifier from the great diversity of versions of the [ISO10646-1] standard for which numerous amendments are still under discussion.

To exclude the control characters we write:

```
C0 ::= UniversalString (FROM ({0,0,0,0}..{0,0,0,31}))
C1 ::= UniversalString (FROM ({0,0,0,128}..{0,0,0,159}))
VanillaUniversalString ::= UniversalString
                             (FROM (ALL EXCEPT (C0|C1)))
```

where the character set `C0` is presented on page 177.

Not to jeopardize the implementation of a specification and avoid indicating literally the character subset that is supported, we can constrain the `UniversalString` type with a parameter of the abstract syntax (see Section 17.3 on page 389); it can dynamically manage the character subsets available for a specific implementation:

```
ISO10646String{UniversalString:ImplementationSubset,
               UniversalString:ImplementationLevel} ::=
  UniversalString (FROM ((ImplementationSubset UNION
    BasicLatin) INTERSECTION ImplementationLevel)
    !characterSetProblem)
characterSetProblem INTEGER ::= 1
```

Note the particular use of the `characterSetProblem` exception to trigger a specific operation on the receiving application's side in case it can only decode part of the set defined by the sending application.

This is an example of specialization of the parameterized type above:

```
MyLevel2String ::= ISO10646String{
  {HebrewExtended UNION Hiragana}, {Level2} }
```

We shall see in Chapter 14 when we come to presentation context switching types that the Presentation layer's negotiation mechanism to dynamically indicate what character subset is taken into account using the `CHARACTER STRING` type. This mechanism makes the abstract syntax independent from all character string types.

11.10.2 Reference Manual

Since the types `UniversalString`, `BMPString` and `UTF8String` use a particular value notation, we describe it here for simplicity's sake. We shall nevertheless repeat this description in the summary of Section 11.13 on page 192.

Type notation

⟨1⟩ The `UniversalString` type has tag no. 28 of class `UNIVERSAL`. This type appeared in ASN.1:1994.

⟨2⟩ A `UniversalString` comprises the characters defined in the [ISO10646-1] standard or in [Uni96]. It is a set of fixed-length characters: each character of the `UniversalString` type encoded on 4 bytes has only one possible interpretation.

⟨3⟩ The use of the `UniversalString` type with no subtype contraint is not recommended because no implementation can fully conform to it.

⟨4⟩ The `BMPString` type has tag no. 30 of class `UNIVERSAL`. This type appeared in ASN.1:1994.

⟨5⟩ The `BMPString` type's alphabet is a subset of the `UniversalString` type's alphabet. Every character of a `BMPString` is encoded on 2 octets according to the code given in the [ISO10646-1] standard.

⟨6⟩ The `UTF8String` type has tag no. 12 of class `UNIVERSAL`. This type appeared in ASN.1:1997.

⟨7⟩ The strings of type `UTF8String` are the same as those of type `UniversalString`. The `UTF8String` type is therefore equivalent to the `UniversalString` type from the abstract syntax viewpoint and one can be indifferently used for the other. However, as the tags of class `UNIVERSAL` are different, their encoding differ (see next rule).

⟨8⟩ Every character of a `UTF8String` is encoded on a number of bytes as small as possible in accordance with the [ISO10646-1Amd2] standard (see Table 11.3 on page 191).

Value notation

$$\begin{aligned} \textit{RestrictedCharacterStringValue} \rightarrow\ & \textsf{cstring} \\ |\ & \textit{CharacterStringList} \\ |\ & \textit{Quadruple} \\ |\ & \underline{\textit{Tuple}} \end{aligned}$$

$$\textit{CharacterStringList} \rightarrow \text{“\{”}\ \textit{CharsDefn}\ \text{“,”}\ \cdots^{+}\ \text{“\}”}$$

$$\begin{aligned} \textit{CharsDefn} \rightarrow\ & \textsf{cstring} \\ |\ & \textit{Quadruple} \\ |\ & \underline{\textit{Tuple}} \\ |\ & \underline{\textit{DefinedValue}} \end{aligned}$$

⟨9⟩ All the characters of cstring must belong to the alphabet associated with the governing type.

⟨10⟩ The cstring can be ambiguous if in the type's alphabet, the same glyph (graphical symbol) is used to represent several characters. In this case, it is recommended to import one of the character references defined in the standardized module `ASN1-CHARACTER-MODULE` (see rule ⟨52⟩ on page 197) and to use it in the *DefinedValue* alternative.
⟨11⟩ *DefinedValue* must be a character string of type `IA5String`, `UniversalString`, `BMPString` or `UTF8String`. This type must be compatible with that of the string being defined (see Section 11.14 on page 197).

Quadruple	→	"{" *Group* "," *Plane* "," *Row* "," *Cell* "}"
Group	→	number
Plane	→	number
Row	→	number
Cell	→	number

⟨12⟩ *Group*, *Plane*, *Row* and *Cell* refer to an abstract character in the encoding space of the *Universal multiple-octet coded Character Set* (*UCS*) defined in [ISO10646-1] and [Uni96].
⟨13⟩ The use of *Quadruple* for defining abstract strings does not influence the encoding size of the characters.
⟨14⟩ In *Group*, number must be smaller than or equal 127.
⟨15⟩ In *Plane*, *Row* and *Cell*, number must be smaller than or equal 255.
⟨16⟩ It is impossible to use a reference (*DefinedValue*) to a value of type `INTEGER` instead of the lexeme number.
⟨17⟩ For the types `UniversalString` and `UTF8String`, the canonical order of the characters (see rule ⟨1⟩ on page 265) is given by:
$256^3 \times Group + 256^2 \times Plane + 256 \times Row + Cell.$
⟨18⟩ For the `BMPString` type, the canonical order of the characters (see rule ⟨1⟩ on page 265) is given by: $256 \times Row + Cell.$

11.11 The `BMPString` type

Today, only part of the 65,536 first cells of the [ISO10646-1] standard is allocated (see previous section) and it is useless to encode each character on four bytes since the first two bytes are systematically null. Indeed, all these 65,536 cells belong to the first plane (group 0, plane 0) called Basic Multilingual Plane (BMP). This encoding on two bytes is called UCS-2.

In order to offer a more efficient encoding for the characters of this plan without compelling the specifier to constrain the `UniversalString` type, but also to allow the basic encoding rules (BER) to provide this encoding, the ASN.1 standard has defined since 1994 the type `BMPString` that has its own tag (`30`) of class `UNIVERSAL`.

The use of this type is now widespread and should be encouraged, instead of the `UniversalString` type, to obtain a more compact encoding. The ASN.1 standard defines it as:

```
BMPString ::=
  [UNIVERSAL 30] IMPLICIT UniversalString (Bmp)
```

where the set `Bmp` is defined in the standardized module `ASN1-CHARACTER-MODULE` (see rule ⟨52⟩ on page 197).

In an ASN.1 module, the BMP plane cell positions are denoted by a quadruple as in the `UniversalString` type (see Section 11.10.2 on page 187), even if the first two coordinates equal zero[24]:

```
latinCapitalLetterA BMPString ::= {0,0,0,65}
greekCapitalLetterSigma BMPString ::= {0,0,3,145}
```

11.12 The `UTF8String` type

The generality of the `UniversalString` and `BMPString` types is appealing indeed but unfortunately their encoding is not compatible with all the existing implementations and protocols. Suppose an old e-mail system receives a message of the [ISO10646-1] format: it would be helpful if the system could decode at least the ASCII alphabet and discard the characters unknown to it.

This is the reason why [ISO10646-1Amd2] offers a variable format which encode ASCII characters on one octet (7 bits in fact) accordingly with the IA5 alphabet of Table 11.2 on page 178 and the others in a sequence of two to six octets (the high-order bit of the first octet equals `1` in this case), as partially described in Table 11.3 on the next page.

This encoding format is called 'UCS Transformation Format, 8-bit form' (UTF-8) and the corresponding ASN.1 type is `UTF8String`[25]. For the abstract syntax viewpoint, it is equivalent to the `UniversalString` type and therefore includes has the same alphabet. The strings of

[24]The notation *Tuple* introduced on page 197 applies only on an `IA5String`.

[25]It was introduced by means of an amendment on ASN.1:1994 and is now part of the ASN.1:1997 standard.

Value	**1st octet**	**2nd octet**	**3rd octet**
`000000000xxxxxxx`	`0xxxxxxx`		
`00000yyyyyxxxxxx`	`110yyyyy`	`10xxxxxx`	
`zzzzyyyyyyxxxxxx`	`1110zzzz`	`10yyyyyy`	`10xxxxxx`

Table 11.3: UTF-8 encoding bit distribution for the Basic Multilingual Plane [Uni96, Section A.2]

type `UTF8String` are defined in ASN.1 as character strings of type `UniversalString`:

```
MyModule DEFINITIONS ::=
BEGIN
IMPORTS latinCapitalLetterA, greekCapitalLetterAlpha
  FROM ASN1-CHARACTER-MODULE {joint-iso-itu-t
    asn1(1) specification(0) modules(0) iso10646(0)};

my-string UTF8String ::= { "This is a capital A: ",
  latinCapitalLetterA, ", and a capital alpha: ",
  greekCapitalLetterAlpha, "; spot the difference!" }
END
```

But the `UTF8String` type has its own tag of class `UNIVERSAL` in order to be able to use its specific encoding. The UTF-8 encoding may look like a backward step compared to the [ISO10646-1] standard (and its UCS-4 encoding) but it offers very interesting features [RFC2279] [Uni96, Section A.2]:

- the bi-directional conversion between the canonical encoding UCS-4 and UTF-8 is simpler and faster (a conversion software written in *C* language is given in [Uni96, Section A.2]);

- the character boundaries can be very easily detected in a byte stream (using the first byte);

- the lexicographical order of the UCS-4 encoded strings is preserved;

- fast search algorithms like Boyer-Moore's[26] can be applied to UTF-8 encoded strings;

[26] http://sunburn.informatik.uni-tuebingen.de/~buehler/BM/BM.html

- the UTF-8 encoding of a character string can be statistically recognized by a receiver with a very low error rate;
- it is a reasonably compact encoding (in the best case, a 25% gain can be obtained compared to the UCS-4 encoding; contrarywise, in the worst case, the UTF-8 encoding is longer).

The UTF-8 encoding format should be supported by all the web servers and all the browsers using XML[27], the forthcoming language of the web. It is the character string type recommended for specifications that are aimed at being used internationally.

11.13 Reference Manual

Type notation

CharacterStringType → *RestrictedCharacterStringType*
| *UnrestrictedCharacterStringType*

RestrictedCharacterStringType → `BMPString`

```
| GeneralString    | GraphicString
| IA5String        | ISO646String
| NumericString    | PrintableString
| TeletexString    | T61String
| UniversalString  | UTF8String
| VideotexString   | VisibleString
```

Type `BMPString`

⟨1⟩ The `BMPString` type has tag no. 30 of class `UNIVERSAL`. It first appeared in ASN.1:1994.
⟨2⟩ The `BMPString` character set is a subset of the `UniversalString` character set. Every character of a `BMPString` is encoded on 2 octets according to the code given in the [ISO10646-1] standard.
⟨3⟩ The canonical order (see rule ⟨1⟩ on page 265) of the characters for the `BMPString` type is defined in rule ⟨47⟩ on page 196.

Type `GeneralString`

⟨4⟩ The `GeneralString` type has tag no. 27 of class `UNIVERSAL`. It groups all the character strings of type `GraphicString`, together with the control characters.

[27] http://www.w3.org/XML/

⟨5⟩ The use of an unconstrained `GeneralString` type is not recommended in practice since this is generally impossible to conform to (see rule ⟨9⟩ on the current page).

Type `GraphicString`

⟨6⟩ The `GraphicString` type has tag no. 25 of class `UNIVERSAL`.

⟨7⟩ The size of `GraphicString` characters encoding may vary; it can convey characters of any character set thanks to the escape mechanism, which describes how the following characters should be interpreted (Latin, Greek... alphabets), but it contains no control character (see the `GeneralString` type). These characters are defined in the document 'International Register of Coded Character Sets to be used with Escape Sequences' [ISOReg]. It is possible to create new character sets, and to register and use them using a specific escape sequence.

⟨8⟩ Whenever possible, one of the types `UniversalString`, `BMPString` or `UTF8String` should be preferred (see rule ⟨22⟩ on the following page).

⟨9⟩ The use of unconstrained `GraphicString` or `GeneralString` types is not recommended because, in practice, this is impossible to conform to. Generally speaking, compilers do not check values of any of these types. This would imply to take into account the escape mechanism to determine the number of characters to be analyzed, which may prove computationally intractable.

⟨10⟩ The `ObjectDescriptor` type is a particular use of the `GraphicString` type (see rule ⟨2⟩ on page 199).

Type `IA5String`

⟨11⟩ The type `IA5String` has tag no. 22 of class `UNIVERSAL`. This type supports no escape character.

⟨12⟩ The character canonical order for `IA5String` is the character definition order given in the [ISO646] standard (see Table 11.2 on page 178).

Type `ISO646String`

⟨13⟩ The type `ISO646String` is equivalent to `VisibleString`. It has the same tag no. 26 of class `UNIVERSAL` (see rules ⟨32⟩ and ⟨33⟩ on page 195).

Type `NumericString`

⟨14⟩ The `NumericString` type has tag no. 18 of class `UNIVERSAL`.

⟨15⟩ A `NumericString` consists of the character `space` and of those from "`0`" to "`9`". These characters are classified in their canonical order.

Type `PrintableString`

⟨16⟩ The type `PrintableString` has tag no. 19 of class `UNIVERSAL`.

⟨17⟩ A `PrintableString` consists of the characters `space`, "`'`", "`(`", "`)`", "`+`", "`,`", "`-`", "`.`", "`/`", "`0`" to "`9`", "`:`", "`=`", "`?`", "`A`" to "`Z`" and "`a`" to "`z`". These characters are classified in their canonical order.

Types `TeletexString` **and** `T61String`

⟨18⟩ The `TeletexString` type has tag no. 20 of class `UNIVERSAL`.

⟨19⟩ The character sets allowed for the `TeletexString` type are summarized in Table 11.1 on page 175. They include the escape character.

⟨20⟩ The `T61String` type is equivalent to `TeletexString`. It has the same tag no. 20 of class `UNIVERSAL`.

Type `UniversalString`

⟨21⟩ The `UniversalString` type has tag no. 28 of class `UNIVERSAL`. It appeared in ASN.1:1994.

⟨22⟩ A `UniversalString` is made of the characters defined in the standards [ISO10646-1] or [Uni96]. It includes characters of fixed length: every character of type `UniversalString` described on 4 octets has only one possible interpretation.

⟨23⟩ The characters canonical order for the type `UniversalString` is defined in rule ⟨46⟩ on page 196.

⟨24⟩ The use of type `UniversalString` with no subtype contraint is not recommended because no implementation can fully conform to it.

Type `UTF8String`

⟨25⟩ The type `UTF8String` has tag no. 12 of class `UNIVERSAL`. It appeared in ASN.1:1997.

⟨26⟩ The strings of type `UTF8String` are the same as those of type `UniversalString`. The `UTF8String` type is therefore equivalent to `UniversalString` from the abstract syntax viewpoint: one can indifferently be used for the other. Still, the tags of class `UNIVERSAL` are different so their encoding are different (see next rule).

⟨27⟩ Every character of a `UTF8String` is encoded on a number of bytes as small as possible according to the [ISO10646-1Amd2] standard (see Table 11.3 on page 191).

⟨28⟩ The canonical order of characters for `UTF8String` is defined in rule ⟨46⟩ on page 196.

Type `VideotexString`

⟨29⟩ The type `VideotexString` has tag no. 21 of class `UNIVERSAL`.
⟨30⟩ The character sets allowed for the `VideotexString` type are summarized in Table 11.1 on page 175. They include the escape character.

Type `VisibleString`

⟨31⟩ The `VisibleString` type has tag no. 26 of class `UNIVERSAL`. It is equivalent to the `ISO646String` type.
⟨32⟩ The `VisibleString` type includes the same characters as `IA5String` except the first 32 characters (LF, CR, HT...) and the delete character. The escape characters are not allowed (see Table 11.1 on page 175).
⟨33⟩ The canonical order of characters for `VisibleString` is the definition order of these characters in the [ISO646] standard.
⟨34⟩ The types `GeneralizedTime` (rule ⟨2⟩ on page 201) and `UTCTime` (rule ⟨2⟩ on page 202) are a particular use of the `VisibleString` type.

⟨35⟩ All the character string types can be constrained by a single value (production *SingleValue* on page 261), by type inclusion (production *ContainedSubtype* on page 263), by size (production *SizeConstraint* on page 267) or by alphabet restriction (production *PermittedAlphabet* on page 269).

Value notation

CharacterStringValue → *RestrictedCharacterStringValue*
| *UnrestrictedCharacterStringValue*

RestrictedCharacterStringValue → cstring
| *CharacterStringList*
| *Quadruple*
| *Tuple*

⟨36⟩ All the characters of a cstring must belong to the alphabet associated with the governing type (see Table 11.1 on page 175, see also the rules ⟨15⟩ and ⟨17⟩ on the preceding page, ⟨2⟩ on page 199, ⟨2⟩ on page 201 and ⟨2⟩ on page 202).
⟨37⟩ The character string cstring can be ambiguous if the same glyph (graphical symbol) is used to represent several characters. In this case, when dealing with the types `IA5String`, `UniversalString`, `UTF8String` and `BMPString`, the use of the alternative *DefinedValue*

for importing character references defined in the standardized module `ASN1-CHARACTER-MODULE` is recommended (see rule ⟨52⟩ on the next page).

CharacterStringList → "{" *CharsDefn* "," $\cdots^{+}$ "}"

⟨38⟩ *CharacterStringList* can be used only for defining values of type `IA5String`, `UniversalString`, `UTF8String` or `BMPString`.

CharsDefn → cstring
| *Quadruple*
| *Tuple*
| *DefinedValue*

⟨39⟩ *DefinedValue* must denote a character string of type `IA5String`, `UniversalString`, `BMPString` or `UTF8String`. This type should be compatible with that of the type which is being defined (see Section 11.14 on the next page).

Quadruple → "{" *Group* "," *Plane* "," *Row* "," *Cell* "}"
Group → number
Plane → number
Row → number
Cell → number

⟨40⟩ *Quadruple* can be used only for denoting character strings (more precisely 'one-character strings') of type `UniversalString`, `BMPString` or `UTF8String`.
⟨41⟩ The use of *Quadruple* for defining abstract strings does not influence the encoding size of the characters.
⟨42⟩ *Group*, *Plane*, *Row* and *Cell* refer to an abstract character in the encoding space of the *Universal multiple-octet coded Character Set* (*UCS*) defined in [ISO10646-1] or [Uni96].
⟨43⟩ In *Group*, number should be smaller or equal to 127.
⟨44⟩ In *Plane*, *Row* and *Cell*, number should be smaller or equal to 255.
⟨45⟩ It is impossible to use a reference (*DefinedValue*) to a value of type `INTEGER` instead of the lexeme number.
⟨46⟩ For the types `UniversalString` and `UTF8String`, the canonical order of the characters (see rule ⟨1⟩ on page 265) is given by the formula: $256^3 \times Group + 256^2 \times Plane + 256 \times Row + Cell$.
⟨47⟩ For the `BMPString` type, the canonical order of the characters (see rule ⟨1⟩ on page 265) is given by the formula: $256 \times Row + Cell$.

$$\begin{array}{lll} \textit{Tuple} & \rightarrow & \text{“\{”}\ \textit{TableColumn}\ \text{“,”}\ \textit{TableRow}\ \text{“\}”} \\ \textit{TableColumn} & \rightarrow & \mathsf{number} \\ \textit{TableRow} & \rightarrow & \mathsf{number} \end{array}$$

⟨48⟩ *Tuple* can be used only for denoting characters of type `IA5String` according to Table 1 in the [ISO2022] standard or to Table 11.1 on page 175.
⟨49⟩ *TableColumn* should take a value between 0 and 7.
⟨50⟩ *TableRow* should take a value between 0 and 15.
⟨51⟩ For the moment, it is impossible to use a reference (*DefinedValue*) to a value of type `INTEGER` instead of the lexeme number.
⟨52⟩ The standardized module `ASN1-CHARACTER-MODULE`[28] (whose object identifier is {`joint-iso-itu-t(2) specification(0) modules(0) iso10646(0)`}) defines value references of type `BMPString` for each character and each character set of the [ISO10646-1] standard; it also defines values of type `IA5String` for the control characters of the IA5 alphabet (columns in yellow in Table 11.2 on page 178) [ISO8824-1, clause 37.1].

11.14 Character string type compatibility

Before the semantic model [ISO8824-1Amd2] of June 1999 (see Section 9.4 on page 121), the ASN.1 standard defined no compatibility between character string types: it was impossible to reference a character string of a given type in another's definition since the meaning of some characters depended on the character string type[29]. The only possible conversion was that which still exists between the types `UniversalString`, `BMPString` and `UTF8String`.

It is worthwhile noticing that this compatibility applies only on the abstract syntax level for making the character string definition easier (indeed, since the ASN.1 abstract values are never encoded, this compatibility does not influence the encoding).

The semantic model now gives a correspondence between some character string alphabets in order to define a compatibility relation (see Section 9.4 on page 121). The types involved are `IA5String`, `VisibleString` (or `ISO646String`), `NumericString`, `PrintableString`, `UniversalString`,

[28] http://asn1.elibel.tm.fr/en/standards/ASN1-CHARACTER-MODULE.asn
[29] The programs “C3 System for conversion character sets” (http://www.nada.kth.se/i18n/c3/) of the TERENA association and François Pinard's ‘recode’ (http://www.iro.umontreal.ca/~pinard/recode/) can carry out conversions from one type to another.

`BMPString` and `UTF8String`. However, no compatibility is defined between the types `GeneralString`, `GraphicString`, `TeletexString` (or `T61String`), `VideotexString` and `CHARACTER STRING`.

Nevertheless, there still exists characters that have the same glyph so it is recommended to subtype the corresponding type to prevent from any reading ambiguity as in the example below:

```
a1 UniversalString(BasicLatin1) ::= "A"
a2 UniversalString(Cyrillic) ::= "A"
```

11.15 The `ObjectDescriptor` type

Having discussed in great length the various character string types, we now present three types called "useful" in the standard! They are actually derived from the `GraphicString` and `VisibleString` types that we have just defined.

11.15.1 User's Guide

The type `ObjectDescriptor`, formally defined by:

```
ObjectDescriptor ::=
  [UNIVERSAL 7] IMPLICIT GraphicString
```

was introduced simultaneously with the `OBJECT IDENTIFIER` type (see Section 10.8 on page 153). The [ISO9834-1] standard recommends that a character string of this type be associated with every new object to be registered in the universal registration tree so that the associated object identifier could be commented. In fact, we have already encountered in Sections 11.2 and 11.3, the object descriptors for the types `NumericString` and `PrintableString` respectively:

```
descr-NumericString ObjectDescriptor ::=
   "NumericString character abstract syntax"
descr-PrintableString ObjectDescriptor ::=
   "PrintableString character abstract syntax"
```

And this is the object descriptor for the basic encoding rules (BER):

```
descr-BER ObjectDescriptor ::=
   "Basic Encoding of a single ASN.1 type"
```

The descriptors are assumed universally unique like the object identifiers, but nothing ensures that because no specific registration procedures are associated with these descriptors. Besides, as the

`GraphicString` type is too general and its size unlimited (by a `SIZE` constraint), the `ObjectDescriptor` type is not properly supported by compilers. Moreover, the object identifiers, in case they include all their identifiers (lexical tokens beginning with a lower-case letter), are generally readable and clear enough. This type is therefore hardly ever used.

11.15.2 Reference Manual

```
UsefulType → GeneralizedTime
           | UTCTime
           | ObjectDescriptor
```

⟨1⟩ The type `ObjectDescriptor` has tag no. 7 of class `UNIVERSAL`.
⟨2⟩ A value of type `ObjectDescriptor` is actually a value of type `GraphicString` (see rule ⟨7⟩ on page 193).
⟨3⟩ According to the [ISO9834-1] standard about the registration procedures, a value of type `ObjectDescriptor` must be associated with every defined object identifier (see Section 10.8 on page 153).

11.16 The `GeneralizedTime` type

11.16.1 User's Guide

This type makes it possible to model a date and an hour by means of a character string conforming to the [ISO8601] standard, which specifies in particular the representation of dates A.D. as well as the hour format.

This format can remove interpretation ambiguities of a notation such as "5/12", which means "5th of December" in France and "12th of May" in Anglo-Saxon countries. A value of type `GeneralizedTime` is therefore made of:

1. the calendar date with four digits for the year, two for the month and two for an ordinal number standing for the day;

2. the time with an hour, minute or second precision (or even fractions of a second) according to the precision of the communicating application;

3. the indication of a possible time lag (the default is the local hour): if it is followed by the letter '`Z`'[30], it denotes the universal time

[30] As the "Zulu" code used all around the world by pilots.

coordinate (UTC)[31]; otherwise, the hour is followed by a positive or negative time lag expressed in hours and minutes whether it is ahead or behind the UTC.

These are a few values that can represent the date '28th of May 1998, 14:29:05.1':

```
-- local hour:
now GeneralizedTime        ::= "19980528142905.1"
now GeneralizedTime        ::= "19980528142905,1"
-- UTC hour:
utc-time GeneralizedTime   ::= "1998052814Z"
utc-time GeneralizedTime   ::= "199805281429Z"
-- ahead of UTC:
ahead-time GeneralizedTime ::= "199805281629+0200"
```

The `GeneralizedTime` type is for example used in the Kerberos protocol (see on page 87) to represent the timestamps:

```
KerberosTime ::= GeneralizedTime
  -- specify a UTC time at a second precision[32]
```

but also in the CMIP protocol [ISO9596-1] (see Section 23.3 on page 482) to return the deletion time of a managed object:

```
DeleteResult ::= SEQUENCE {
  managedObjectClass    ObjectClass OPTIONAL,
  managedObjectInstance ObjectInstance OPTIONAL,
  currentTime           [5] IMPLICIT GeneralizedTime
                                     OPTIONAL }
```

In fact the `GeneralizedTime` type is aimed at international applications for which the local time is not appropriate. In the general case, the specifier may define a particular type for denoting time in order to make the interpretation easier for the programmer of the communicating application.

11.16.2 Reference Manual

UsefulType → `GeneralizedTime`
| `UTCTime`
| `ObjectDescriptor`

[31]It is also the new name for the Greenwich meridian time, according to the recommendation of the 'General Conference on Weights and Measures' in 1975.

[32]This comment could appear in a user-defined constraint denoted by the keywords `CONSTRAINED BY` (see Section 13.13 on page 294).

⟨1⟩ The type `GeneralizedTime` has tag no. 24 of class `UNIVERSAL`.
⟨2⟩ A value of type `GeneralizedTime` is a character string of type `VisibleString` with one of the following lexical restrictions:

a) a string of the form “`AAAAMMJJhh`[`mm`[`ss`[(`.`|`,`)`ffff`]]]”[33] standing for a local time, four digits for the year, two for the month, two for the day and two for the hour (the value 24 is forbidden), followed by two digits for the minutes and two for the seconds if required, then a dot (or a comma), and a number for the fractions of second (the maximum precision depends on the application) [ISO8601][34]; or

b) a string of *a*) followed by the letter “`Z`” (which would denote a UTC time); or

c) a string of *a*) followed by a string “(`+`|`-`)`hh`[`mm`]” (then the UTC time is the difference between the string of *a*) and the second string).

⟨3⟩ ASN.1 provides no syntactical means of constraining the `GeneralizedTime` type to make times respect one of the restrictions described in the previous rule.
⟨4⟩ It is recommended not to apply subtype constraints on the `GeneralizedTime` type (in particular, the semantic of the `SIZE` constraint is not defined).
⟨5⟩ In the ASN.1:1994/97 standards, the `GeneralizedTime` type cannot to specify the leap second (`23:59:60`), which is sometimes inserted after the last second (`23:59:59`) of the last day of June or December in order to reduce the delay with the international atomic time[35]. It is therefore better to avoid using values just before midnight. This lack is going to be corrected.

[33]The square brackets indicate that the expression they contain is optional; the round brackets and the vertical bar gives a choice between two alternatives.
[34]The precision may also depend on the modifications of the [ISO8601] standard referenced by ASN.1.
[35]Information on time description can be found at http://www.bldrdoc.gov/timefreq/service/nts.htm for example.

11.17 The `UTCTime` type

11.17.1 User's Guide

In case the flexibility offered by the various formats of the `GeneralizedTime` is not necessary, one may use the `UTCTime` type whose (more restricted) format is the following:

1. the calendar date with two digits for the year, two for the month and two for the day; and
2. the hour, minutes and seconds; and
3. either the capital letter '`Z`' to indicate that the time is the UTC or a positive or negative delay with respect to the UTC.

These are some strings conforming to the `UTCTime` format; they model the date '28th of May, 1998 14:29:05':

```
now UTCTime  ::= "9805281429Z"
now UTCTime  ::= "980528142905Z"
delayed-hour UTCTime ::= "9805281629+0200"
```

Contrary to the `GeneralizedTime` type, the `UTCTime` type use only two digits for the year and can therefore be very much concerned with the now famous millennium bug! Quite rightly, the ASN.1 standard gives no directions as to how this can be solved: indeed, it is just a matter of interpretation of the two digits of the dates by the applications and not a problem of transfer for these two digits. Two working groups have nonetheless suggested an interpretation: these are reproduced in the rules ⟨6⟩ and ⟨7⟩ on the next page. But it might be preferable, when possible, to operate a migration towards the `GeneralizedTime` type.

11.17.2 Reference Manual

UsefulType → `GeneralizedTime`
| `UTCTime`
| `ObjectDescriptor`

⟨1⟩ The `UTCTime` type has tag no. 23 of class `UNIVERSAL`.
⟨2⟩ A value of type `UTCTime` is a value of type `VisibleString` with one of the following lexical restrictions:

a) a string of the form "`AAMMJJhhmm[ss]Z`", which represents the UTC time with 2 digits for the year, 2 for the month (from `01` to `12`), 2 for the day (from `01` to `31`), 2 for the hour (from `00` to `23`), 2 for the minutes (from `00` to `59`), possibly followed by 2 digits for the seconds (from `00` to `59`, see rule ⟨5⟩); this string ends with the capital letter "`Z`"; or

b) the string "`AAMMJJhhmm[ss]`" of *a*) (without the letter "`Z`") followed by a string of the form "`(+|-)hhmm`" (the UTC is then given by the difference between the string in *a*) and the second string).

⟨3⟩ ASN.1 provides no syntactical means of constraining the `UTCTime` type to make times respect one of the restrictions described in the previous rule.

⟨4⟩ It is recommended not to apply subtype constraints on the `UTCTime` type (in particular, the semantic of the `SIZE` constraint is not defined).

⟨5⟩ In the ASN.1:1994/97 standards, the `UTCTime` type cannot to specify a leap second (`23:59:60`) which is sometimes inserted after the last second (`23:59:59`) of the last day of June or December in order to reduce the delay with the international atomic time[36]. It is therefore better to avoid using values just before midnight. A way to correct this lack is being considered.

⟨6⟩ The ITU-T working group on the X.500 directory and the ITU-T Q22/11 question (intelligent networks protocols) propose that an application should be able to sort the `UTCTime` dates and interpret them in the 1950-2049 interval, which would mean, for example, that the value "`0105281429Z`" corresponds to the 28th of May 2001.

⟨7⟩ The ITU-T working group on the X.400 e-mail propose that dates from 10 years backwards and 40 years ahead should be interpreted with respect to the 20th century and that the interpretation of the other years should be left to the programmer's interpretation (for example, for an application that is running in 1998, the values from "`88`" to "`99`" stand for the years from 1988 to 1999, the values from "`00`" to "`38`" are the years from 2000 to 2038, and the interpretation of the values from "`39`" to "`87`" depend on the implementation); this proposition of 'shifting window' is well-adapted for the MHS e-mail systems that manage dates in the past (dates of old messages stored or forwarded) as well as in the future (expiry date).

[36]Information on time description can be found at http://www.bldrdoc.gov/timefreq/service/nts.htm for example.

Chapter 12

Constructed types, tagging, extensibility rules

Contents

As she worked, some ideas began to occur to her about the blocks.

They are all easy to assemble, she thought. Even though they are all different, they all fit together. They are also unbreakable. [...] The best thing about them was that with Lego she could construct any kind of object. And then she could separate the blocks and construct something new.

What more could one ask of a toy?

Jostein Gaarder, *Sophie's World.*

Having described almost all ASN.1 basic types (the obsolete `ANY` type will be introduced at the end of this chapter and the presentation context switching types will be discussed in Chapter 14), we now introduce the 'constructed types'[1], used for defining types whose values are structured.

Unfortunately, semantic rules of some constructors depend on the concept of tagging which have to be introduced in the first place... using type constructors for this purpose! In their description, we have tried to avoid, as much as possible, references to subsequent sections to prevent from using a notion before defining it; these can nonetheless be tolerated when necessary for understanding ASN.1 concepts and when these cross-references merely reflect the notions' interdependence.

We conclude this chapter with the concept of extensibility, which makes possible the compatibility of successive versions of the same specification.

12.1 Tagging

When a value is transmitted, all ambiguities must be avoided so that it could be properly decoded and interpreted by the receiving application. In particular, the uniqueness of every type of value that may potentially be transferred should be ensured to allow the receiver to determine exactly the type of data it receives. The syntactical restriction according to which two type definitions cannot have the same reference (the same name) is of no help since these abstract references are not encoded.

Historically[2], the first encoding rules associated with ASN.1 were the basic encoding rules (BER). As described in Chapter 18, the BER transfer syntax has the form of a triplet ⟨type, length, value⟩ or TLV where type (also called tag) is a code that identifies unambiguously the transmitted data type, length is the number of bytes necessary for encoding the value and value is the actual ASN.1 value encoding. In ASN.1, this 'unique identification code' of every type is called a *tag*.

[1]We shall sometimes use this term to stress the fact that a 'constructed type' invokes other ASN.1 types that may be basic types, constrained types or even type constructors themselves (like Lego). Otherwise said, a type constructor without its own arguments, which are types strictly speaking, does not denote a set of ASN.1 values. Still, from ASN.1 viewpoint, a type constructor is conceptually a type of its own, and will, therefore, have specific encoding rules.

[2]The notion of 'tagging' already existed in Xerox's Courier (see on page 60) introduced at the time for making the structured type's extensibility easier. We shall see in Section 12.9 on page 244 that the extension marker "..." has had the same function since 1994 without the specifier's having to care about tagging.

We shall see in Chapter 20 that the packed encoding rules (PER) are not based on the same format and do not use the notion of tag. The specifier can get round the problem of tagging (whatever the encoding rules) by inserting the clause `AUTOMATIC TAGS` in the module header (see Section 12.1.3 on page 213).

12.1.1 Tags and tagging classes

A tag is associated (very often by default) with every type of an ASN.1 module. It can be seen as a couple ⟨tagging class, number⟩ where number is a positive integer. There exists four tagging classes: `UNIVERSAL`, context-specific, `APPLICATION` and `PRIVATE`. These make it possible to define the scope (or context) where every tag must be unique. In a given context, two tags are considered to be different if they are of different classes or if their respective numbers are different.

Before introducing the problem of tagging in a few examples, it should be underlined that when the specifier has to tag explicitly a type, the tagging class and the tag are written in square brackets before the type:

```
T ::= [5] INTEGER
Afters ::= CHOICE { cheese  [0] IA5String,
                    dessert [1] IA5String }
ClientNumber ::= [APPLICATION 0] NumericString
```

The insertion of a tag before a type does not modify the type's abstract definition (an untagged type and the same type with a tag are isomorphic), nor does it change the definition of values of this type. Indeed, since the abstract values in ASN.1 are never encoded, there is no reason why these should be concerned with tags[3].

The `UNIVERSAL` class is reserved for the ASN.1 standard designers who allocate a tag of this class to all new standardized types (the tags that have been allocated so far are presented in Table 12.1 on page 209). *The specifiers are not allowed to use this class*[4]. The scope uniqueness of

[3]For the same reason, there would be no point in tagging the type of an abstract value as in "`v [1] INTEGER ::= 5`". A tag should systematically appear on the right-hand side of the symbol "`::=`".

[4]In a module, we can come across definitions such as:

```
UTF8String ::= [UNIVERSAL 12] IMPLICIT OCTET STRING
```

used for getting round restrictions of some ASN.1 tools, but such definitions are forbidden and must be rejected by all tools.

the `UNIVERSAL` class is the ASN.1 standard, that is to say more generally: two types defined in the standard cannot have the same default tag.

But in the following choice:

```
Afters ::= CHOICE { cheese  IA5String,
                    dessert IA5String }
```

the tag `[UNIVERSAL 22]` is affected by default to the two alternatives and it is therefore impossible to say at the decoding stage whether the meal was finished with cheese or dessert (but not both!). It is therefore down to the specifier to locally overwrite the default tagging of the two alternatives of the `CHOICE` as follows:

```
Afters ::= CHOICE { cheese  [0] IA5String,
                    dessert [1] IA5String }
```

These tags that seem to refer to no tagging class in particular are actually of *context-specific* class whose scope is that of the type constructor `SEQUENCE`, `SET` or `CHOICE` that includes them (or more precisely the component or alternative list in curly brackets). The context-specific class is the most commonly used and has no keyword to denote it; it is therefore the default tagging class if no keyword appears in the tagging square brackets.

The following example shows the context-specific tagging class that can be used in the scope of a type constructor (it is not recommended to put the `[0]` and `[1]` tags outside the `SET` type):

```
Form ::= SET {
  name             Surname,
  first-name       First-name,
  phone-number [2] NumericString }
Surname    ::= [0] VisibleString
First-name ::= [1] VisibleString
```

The tag numbers of class context-specific can be re-used if the types are constructed one above the other since the tags are not relevant outside the scope of the constructed type:

```
A-possible-type ::= SET {
  integer  [0] CHOICE {
                 a [0] INTEGER,
                 b [1] INTEGER },
  boolean  [1] CHOICE {
                 a [0] BOOLEAN,
                 b [1] BOOLEAN }}
```

`[UNIVERSAL 0]`	reserved for BER (see Chapter 18)
`[UNIVERSAL 1]`	`BOOLEAN`
`[UNIVERSAL 2]`	`INTEGER`
`[UNIVERSAL 3]`	`BIT STRING`
`[UNIVERSAL 4]`	`OCTET STRING`
`[UNIVERSAL 5]`	`NULL`
`[UNIVERSAL 6]`	`OBJECT IDENTIFIER`
`[UNIVERSAL 7]`	`ObjectDescriptor`
`[UNIVERSAL 8]`	`EXTERNAL`, `INSTANCE OF`
`[UNIVERSAL 9]`	`REAL`
`[UNIVERSAL 10]`	`ENUMERATED`
`[UNIVERSAL 11]`	`EMBEDDED PDV`
`[UNIVERSAL 12]`	`UTF8String`
`[UNIVERSAL 13]`	`RELATIVE-OID`
`[UNIVERSAL 14]`	reserved for future use
`[UNIVERSAL 15]`	reserved for future use
`[UNIVERSAL 16]`	`SEQUENCE`, `SEQUENCE OF`
`[UNIVERSAL 17]`	`SET`, `SET OF`
`[UNIVERSAL 18]`	`NumericString`
`[UNIVERSAL 19]`	`PrintableString`
`[UNIVERSAL 20]`	`TeletexString`, `T61String`
`[UNIVERSAL 21]`	`VideotexString`
`[UNIVERSAL 22]`	`IA5String`
`[UNIVERSAL 23]`	`UTCTime`
`[UNIVERSAL 24]`	`GeneralizedTime`
`[UNIVERSAL 25]`	`GraphicString`
`[UNIVERSAL 26]`	`VisibleString`, `ISO646String`
`[UNIVERSAL 27]`	`GeneralString`
`[UNIVERSAL 28]`	`UniversalString`
`[UNIVERSAL 29]`	`CHARACTER STRING`
`[UNIVERSAL 30]`	`BMPString`
`[UNIVERSAL 31]`...	reserved for future use

Table 12.1: Tags of class `UNIVERSAL`

The rules that check the uniqueness of the context-specific class tags within the scope of a constructed type differ from `SEQUENCE` to `SET` or `CHOICE`. They will be described when we come to every one of these constructed types further on in this chapter.

In principle, the classes `UNIVERSAL` and context-specific are sufficient to unambiguously decode the components of a `SEQUENCE` or `SET` type or the alternatives of a `CHOICE` type. At the time the notion of tagging was introduced in the ASN.1 standard, the concept of presentation context was not quite developed and the tags were expected to be used for distinguishing between messages similarly encoded but generated by different applications.

Two other tagging classes were therefore introduced, but since these were not properly explained in the standard, they were erroneously interpreted that lead the ASN.1 standard designers not to recommend their use in 1994.

The standard [X.409] indicated that the `APPLICATION` class would be used "*to define a data type that finds wide, scattered use within a particular application and that must be distinguishable (by means of its [*abstract syntax*]) from all other data types used in the application*". The bottom line idea was to use this class to tag types that would be referenced several times in a specification. As the tag of class `APPLICATION` was by definition different from all the other tags (whatever their class), overwriting the tag of these types would not be necessary.

Such may be the case for the order number given in the case study on page 36:

```
Order-number ::=
  [APPLICATION 0] NumericString (SIZE (12))
```

thereby ensuring that in the type `Order-header` defined in the same chapter, the tag of the `number` component is different from that of the `date` component (even though these two components are of type `NumericString`). Besides, this particular tag also allows re-using the type `Order-number` in the type `Delivery-report`.

Unfortunately, the scope within which the `APPLICATION` class tags must be unique is not clearly defined in the standard: protocols frequently import type definitions for which the condition on distinct tags can hardly be checked. The use of this tagging class is therefore not recommended, all the more since 1994 when the condition on distinct tags for the `APPLICATION` class has been abolished.

For the fourth and last tagging class, the PRIVATE class, the [X.409] standard recommended to "*use a private-use Tagged [*type*] to define a data type that finds use within a particular organization or country and that must be distinguishable (by means of its [*abstract syntax*]) from all other data types used by that organization or country*".

The idea was to use this class for tagging the new components of constructed types (SEQUENCE, SET or CHOICE) when producing a version of a protocol standardized at an international level but locally adapted to a regional area. These components 'private' to a company would therefore never clash with any new components introduced in the standard version.

Unless the data transfer could be limited to the company and the tags of class PRIVATE unique in this scope, a common agreement should be found to transmit data of a protocol using such types. Besides, it is recommended for the company not to design these tags on a long term basis but rather have them standardized so that these PRIVATE class tags could turn into context-specific class tags.

For example, a company may extend its Transport Layer PDU (using an extended component):

```
RejectTPDU ::= SET {
  destRef    [0] Reference,
  yr-tu-nr   [1] TPDUnumber,
  credit     [2] Credit,
  extended   [PRIVATE 0] BOOLEAN DEFAULT FALSE }
```

The use of PRIVATE class tags is not recommended since 1994.

12.1.2 Tagging mode

In the previous section, we have described how a specifier could insert context-specific tags so that the components of a constructed type could be decoded unambiguously:

```
Afters ::= CHOICE { cheese  [0] IA5String,
                    dessert [1] IA5String }
```

Still, we have not dwelt upon what may become of the UNIVERSAL class tag no. 22 associated by default with the IA5String type. With no more information from the specifier and particularly in the context of a BER encoding, the two tags [0] and [UNIVERSAL 22] are encoded if the alternative cheese is chosen. In this case, the tagging is in *explicit mode*.

Even if it may seem pointless to transmit two tags[5] since the unicity of the first one (of context-specific class) is sufficient to ensure an unambiguous decoding, the `UNIVERSAL` class tag proves useful to know the effective type of the `cheese` alternative and allows a monitor that would be unaware of the abstract syntax (i.e. the ASN.1 module) to offer a more user-friendly access to the data when supervising a telecommunication network (such as giving the words `TRUE` or `FALSE` for a value detected as boolean, rather than the raw content of the bytes in hexadecimal notation).

The explicit tagging mode can sometimes make a specification more easily extensible. In this mode, indeed, the type:

```
T1 ::= [0] INTEGER
```

can be replaced by

```
T2 ::= [0] CHOICE { integer INTEGER,
                    real    REAL }
```

while the same BER encoding is preserved for the alternative `integer` of type `T2` as for `T1`.

If the explicit mode encoding is judged too verbose, or too costly, the specifier can use the *implicit mode* inserting (explicitly!) the keyword `IMPLICIT` between the tag and the type:

```
Afters ::= CHOICE { cheese  [0] IMPLICIT IA5String,
                    dessert [1] IMPLICIT IA5String }
```

Since the tagging mode applies to a tag, the `IMPLICIT` marker must appear after a closing square bracket "`]`". We shall see that some types cannot be tagged in implicit mode because it would make decoding non-deterministic: this is the case for the `CHOICE` type, the open type (which has replaced the `ANY` type since 1994), and a parameter that is a type.

The `IMPLICIT` marker proceeds as follows: all the following tags, explicitly mentioned or indirectly reached through a type reference are ignored until the next occurrence (included) of the `UNIVERSAL` class tag (except if the `EXPLICIT` marker is encountered before). So, for the type `T` below:

```
T ::= [1] IMPLICIT T1
T1 ::= [5] IMPLICIT T2
T2 ::= [APPLICATION 0] IMPLICIT INTEGER
```

[5]Remember the tagging mode is relevant if the BER encoding rules (or their derived forms, CER or DER) are used, but need not be considered with PER.

only the tag `[1]` should be encoded. Another way of explaining the concept of implicit tagging is to say that a tag marked `IMPLICIT` *overwrites the tag that follows it* (recursively); hence, for the example above, tag `[1]` overwrites tag `[5]`, which in turn overwrites tag `[APPLICATION 0]` which finally overwrites the default tag `[UNIVERSAL 2]` of the `INTEGER` type.

A type tagged in implicit mode can be decoded only if the receiving application 'knows' the abstract syntax, i.e. the decoder has been generated from the same ASN.1 module as the encoder was (and such is the case most of the time).

12.1.3 Global tagging mode

It might seem complicated or pointless to choose the tagging mode applicable for each and every type, as much as fastidious to insert `IMPLICIT` markers. Some may even say (and they would have a point there!) that such a concept, which comes under the responsibility of the transfer syntax, should never have appeared at the abstract syntax's level, and so much less since the last standardized encoding rules, the PER, hardly use them. Besides, there exists no precedent or equivalent (visible, at least) in programming languages. In fact, adopting the explicit tagging (the most verbose) as the default mode was definitely not the most brilliant idea.

For all these reasons, three global tagging modes[6] can apply to a whole module. The first two modes are self-explicitly named `EXPLICIT TAGS` and `IMPLICIT TAGS`. The global tagging mode is denoted in the module header between the keyword `DEFINITIONS` and the symbol "`::=`":

```
ModuleName DEFINITIONS IMPLICIT TAGS ::=
BEGIN
-- ...
END
```

With the clause `IMPLICIT TAGS`, all the components of types `SEQUENCE`, `SET` and `CHOICE` defined in the module (except the components and alternatives that are followed by an `EXPLICIT` marker, a `CHOICE` type, an open type, or a parameter that is a type) are tagged in implicit mode. The global tagging clause does not affect components of types imported by the `IMPORTS` clause, which remain tagged according to the global tagging

[6]The standard defines these tagging modes as the 'default' modes even though strictly speaking the only default mode is the explicit.

mode of the module where they are defined. The clause `EXPLICIT TAGS` is defined adopting the dual approach.

The following specification:

```
M1 DEFINITIONS EXPLICIT TAGS ::=
BEGIN
T1 ::= CHOICE { cheese  [0] IA5String,
                dessert [1] IA5String }
END

M2 DEFINITIONS IMPLICIT TAGS ::=
BEGIN
IMPORTS T1 FROM M1;
T2 ::= SET { a [0] T1,
             b [1] REAL }
END
```

is therefore equivalent, once applied the tagging modes and importation, to:

```
M1 DEFINITIONS ::=
BEGIN
T1 ::= CHOICE { cheese  [0] EXPLICIT IA5String,
                dessert [1] EXPLICIT IA5String }
END

M2 DEFINITIONS ::=
BEGIN
T1 ::= CHOICE { cheese  [0] EXPLICIT IA5String,
                dessert [1] EXPLICIT IA5String }
T2 ::= SET { a [0] EXPLICIT T1,
             b [1] IMPLICIT REAL }
END
```

where it should be noted that the component `a` of the `T2` type is marked as `EXPLICIT` because it is of type `CHOICE` (see rule ⟨6⟩ on page 216).

All the criticism expressed against tagging mentioned at the beginning of this section, vanished when the `AUTOMATIC TAGS` global tagging mode was introduced in 1994. If a module includes the clause `AUTOMATIC TAGS` in its header, the components of all its structured types (`SEQUENCE`, `SET` or `CHOICE`) are automatically tagged by the compiler starting from 0 by one-increment. By default, every component is tagged in the implicit mode except if it is a `CHOICE` type, an open type or a parameter that is a type. This tagging mechanism is obviously documented in the ASN.1 standard and, as a result, does not depend on the compiler.

Hence, the module:

```
M DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
T ::= SEQUENCE { a INTEGER,
                 b CHOICE { i INTEGER,
                            n NULL },
                 c REAL }
END
```

is equivalent, once applied the automatic tagging, to:

```
M DEFINITIONS ::=
BEGIN
T ::= SEQUENCE {
  a [0] IMPLICIT INTEGER,
  b [1] EXPLICIT CHOICE { i [0] IMPLICIT INTEGER,
                          n [1] IMPLICIT NULL },
  c [2] IMPLICIT REAL }
END
```

If a constructed type contains at least a tag inserted by the specifier (i.e. if it contains square brackets), the automatic tagging mode is 'switched off' for this type. If a `SEQUENCE` or `SET` type contains a `COMPONENTS OF` clause, this very clause is developed before applying the automatic tagging; but the decision of automatic tagging must be taken before development (note that the clause `COMPONENTS OF` may have brought in types that were explicitly tagged by the specifier). Compilers should normally check if such conflicts occur.

It is now recommended to use the `AUTOMATIC TAGS` clause for new specifications[7], a fortiori for abstract syntaxes known to be encoded with the PER since these rules do not transmit the tags.

An automatically tagged module has a data structure that very much looks like any other computing languages or formal notation, and has the advantage of being more readable for a neophyte or anyone who is not quite familiar with the telecommunication jargon.

[7]The old modules that still have the `IMPLICIT TAGS` clause and whose constructed types were numbered 1 by 1 from 0 should also comply with this new clause.

12.1.4 Reference Manual

Type tagging

TaggedType → *Tag* *Type*
  | *Tag* `IMPLICIT` *Type*
  | *Tag* `EXPLICIT` *Type*

Tag → "`[`" *Class* *ClassNumber* "`]`"

⟨1⟩ The PER encoding rules do not use the tagging mode since tags are never transmitted in this case.
⟨2⟩ If a type is tagged, directly or indirectly, in the `EXPLICIT` mode (i.e. the tag is marked `EXPLICIT`, or it is not marked but, either the module contains the clause `EXPLICIT TAGS` in its header, either it contains no global-tagging clause), all the tags preceding the keyword `EXPLICIT` and those which follow (potentially until the next tag marked, directly or indirectly, `IMPLICIT`), including the tag of class `UNIVERSAL` associated by default with the type, are encoded in BER, whatever their class (`UNIVERSAL`, context-specific, `APPLICATION` or `PRIVATE`).
⟨3⟩ If a type is tagged, directly or indirectly, in the `IMPLICIT` mode (i.e. the tag is marked `IMPLICIT`, or it is not marked but the module contains the clause `IMPLICIT TAGS` in its header), only the tags preceding the keyword `IMPLICIT` are transmitted in BER (in particular, the default tag, of class `UNIVERSAL`, is not encoded except if a tag marked, directly or indirectly, `EXPLICIT` is found after the tag marked `IMPLICIT`).
⟨4⟩ For a definition such as `T ::= [1] IMPLICIT [0] EXPLICIT INTEGER`, both the tags `[1]` and `[UNIVERSAL 2]` are transmitted by a BER encoder whereas for `T ::= [1] EXPLICIT [0] IMPLICIT INTEGER`, only the tags `[1]` and `[0]` are transmitted.
⟨5⟩ The tag is, by default, of mode `EXPLICIT` if the module's global tagging mode (production *TagDefault* on page 217) is `EXPLICIT TAGS` or if no global tagging mode is indicated in the module header.
⟨6⟩ The tag is, by default, of `IMPLICIT` mode if the module's global tagging mode is `IMPLICIT TAGS` or `AUTOMATIC TAGS`, except if the *Type* is an untagged `CHOICE`, an untagged open type *ObjectClassFieldType* (which has replaced the `ANY` type since 1994, see on page 347) or an untagged parameter *DummyReference* that is a type (see rule ⟨17⟩ on page 386), for which the tag is always of `EXPLICIT` mode.

Class → `UNIVERSAL`
  | `APPLICATION`
  | `PRIVATE`
  | ϵ

⟨7⟩ The class `UNIVERSAL` can only be used in the ASN.1 standard [ISO8824-1]. Only the ASN.1 standardization group (at ISO and ITU-T) can define new *ClassNumber*s in this `UNIVERSAL` class.
⟨8⟩ In the abstract syntax describing a complete protocol, a tag of class `APPLICATION` must only be assigned to one type. This class must be reserved for types frequently used in the specification or for the PDUs.
⟨9⟩ A standard cannot assign a tag of class `PRIVATE`. The use of `PRIVATE` class tags is specific to a company in order to extend a standardized specification while avoiding clashes with some other company's specification.
⟨10⟩ The use of the classes `APPLICATION` and `PRIVATE` is not recommended since 1994.
⟨11⟩ The condition of distinct context-specific tags (alternative ϵ of the production *Class*) is ensured by the rules ⟨14⟩ on page 224 for the `SEQUENCE` type, ⟨10⟩ on page 228 for the `SET` type and ⟨9⟩ on page 238 for the `CHOICE` type.

ClassNumber → number
  | *DefinedValue*

⟨12⟩ *DefinedValue* is a reference to a positive or null value of type `INTEGER`.
⟨13⟩ In the context of a BER encoding, the *ClassNumber* value should be smaller than or equal 30 so that the tag could be encoded on one byte (see Figure 18.2 on page 396).
⟨14⟩ For the encoding rules that require it, there exists a canonical order for the tags (and their classes) defined in rule ⟨15⟩ on page 228.

Module global tagging mode

ModuleDefinition → *ModuleIdentifier* `DEFINITIONS`
  TagDefault *ExtensionDefault* "`::=`"
  `BEGIN` *ModuleBody* `END`

TagDefault → `EXPLICIT TAGS`
  | `IMPLICIT TAGS`
  | `AUTOMATIC TAGS`
  | ϵ

⟨15⟩ If no clause *TagDefault* appears in the module header, the global tagging mode is `EXPLICIT TAGS` for the whole module.
⟨16⟩ Since 1994, the standard recommends using the `AUTOMATIC TAGS` clause to free the specifier from the distinct tag requirement.
⟨17⟩ If the `AUTOMATIC TAGS` clause is present, all the components of the `SEQUENCE` and `SET` types and all the alternatives of the `CHOICE` types defined in the module are automatically tagged according to the rules⟨6⟩ on page 223, ⟨6⟩ on page 227 and ⟨4⟩ on page 238 respectively.
⟨18⟩ In a type definition, a tag is by default of `IMPLICIT` mode if the *TagDefault* clause of the module is `IMPLICIT TAGS` or `AUTOMATIC TAGS`, except if it is followed by an untagged `CHOICE` type, an untagged open type *ObjectClassFieldType* (which has replaced the `ANY` type since 1994, see on page 347) or an untagged parameter *DummyReference* that is a type (see rule ⟨17⟩ on page 386), for which tags are always of `EXPLICIT` mode.
⟨19⟩ The presence of a *TagDefault* clause does not affect imported types (see rule ⟨18⟩ on page 115).
⟨20⟩ Should a protocol require that values of different types are to be transmitted at any time, every type will have a distinct tag (or else the PDU is necessarily a `CHOICE` of these types).

12.2 The constructor `SEQUENCE`

12.2.1 User's Guide

The constructed type that is the most commonly used by computing languages is that which provides an ordered series of elements (or *components*) of different types. In ASN.1, this structure is introduced by the keyword `SEQUENCE` and every component is denoted by a word beginning with a lower-case letter called an *identifier*:

```
Description ::= SEQUENCE { surname    IA5String,
                           first-name IA5String,
                           age        INTEGER }
```

The type `Description` collects triplets whose first two elements are of type `IA5String` while the third one is of type `INTEGER`. A value of type `Description` is given by:

```
johnny Description ::= { surname    "Smith",
                         first-name "John",
                         age        40 }
```

Identifiers are never encoded since the decoder relies on the tags to identify every component. Nonetheless, the identifiers are not useless: these are formal comments that give the specification's semantics and help the reader understand each component of the structure. Moreover, they can describe abstract values unambiguously[8] as in the example `johnny` above whose surname and first name cannot be confused. The specifier would be therefore well-advised to choose carefully their names. These identifiers must obviously be distinct within a `SEQUENCE` type.

The restrictive order imposed by the `SEQUENCE` may prove interesting, for example, for an X.400 electronic mail application that would sequentially handle the e-mail messages (of a possibly important size) received and should treat the header (envelope) before the message body (content):

```
Message ::= SEQUENCE { envelope MessageTransferEnvelope,
                       content  Content }
```

If we stopped here, the `SEQUENCE` type would be no more than the equivalent of `struct` type to be found in numerous languages. But since it models transmitted data, the information may be unavailable or, under some conditions, not meant to be transmitted or even irrelevant for encoding when taking certain values fixed in advance. For these reasons, ASN.1 provides the markers `OPTIONAL` and `DEFAULT`[9] that are used as follows (whatever the complexity of the type they are associated with):

```
ImprovedDescription ::= SEQUENCE {
  surname    IA5String,
  first-name IA5String OPTIONAL,
  age        INTEGER DEFAULT 40 }
```

[8]These identifiers have been mandatory since 1994. Nothing in the previous standard versions prevented specifiers from writing:

```
T ::= SEQUENCE { INTEGER {a(0), b(1)} OPTIONAL,
                 ENUMERATED {a(0), b(1)} OPTIONAL }
```

for which the abstract value:

```
v T ::= { a }
```

is ambiguous since it is not clear whether it is of type `INTEGER` or `ENUMERATED`. Besides, the distinctive encoding rules (DER), which use the identifiers lexicographical order, could not generate a correct encoder for such a type (see Table 19.2 on page 421).

[9]The clause `DEFAULT` can only appear within a type `SEQUENCE` or `SET` (or in information object class definitions, see Chapter 15). So it is not allowed to define:

```
T ::= SEQUENCE { a U }
U ::= INTEGER DEFAULT 100
```

Concerning the `DEFAULT` marker, the default value should obviously conform to its preceding type. From the application viewpoint, not receiving the component `age` or receiving the default value `40` is equivalent. Besides, some decoders do not even send the default value back to the application when no value was received for this component. The `DEFAULT` clause often applies to the ASN.1 basic types, but its use become more and more common for more complex types[10].

The use of the markers `OPTIONAL` and `DEFAULT` in a `SEQUENCE` type demands a specific rule ('condition on distinct tags') to make sure the decoding is not ambiguous. Let us consider the following untagged type:

```
T ::= SEQUENCE { x INTEGER,
                 y INTEGER OPTIONAL,
                 z INTEGER DEFAULT 0,
                 t INTEGER }
```

and try to transfer the value {`x 1, y 2, t 3`}. For a BER encoding (see Chapter 18), the decoder receives a byte stream that can be *informally* represented as:

T	L	V	T	L	V	T	L	V
2	1	1	2	1	2	2	1	3

In this case, it does not 'know' which components it should give the value `2` and `3` to, because it cannot determine whether the components `y` and `z` were actually transmitted (the order criterion is not sufficient and the three tag fields T include the `UNIVERSAL` class tag 2 for the `INTEGER` type).

The condition on distinct tags, therefore, imposes that for every group of successive optional components, the tags should differ from one another and differ from the mandatory component that follows. The minimal tagging of the type above leads to the following expression:

```
T ::= SEQUENCE { x     INTEGER,         -- [UNIVERSAL 2]
                 y [0] INTEGER OPTIONAL,
                 z [1] INTEGER DEFAULT 0,
                 t     INTEGER }        -- [UNIVERSAL 2]
```

[10]For a DER encoding, we should nevertheless make sure that our compiler can test whether such complex type values are equal (see Table 19.2 on page 421).

Finally, ASN.1 allows the components of a `SEQUENCE` type to be inserted within another `SEQUENCE` type using the clause `COMPONENTS OF`:

```
Registration ::= SEQUENCE {
  COMPONENTS OF Description,
  marital-status ENUMERATED {single, married, divorced,
                             widowed} }
```

Hardly ever used in practice, this syntactic operator re-uses the components of the stated `SEQUENCE` type but does not take into account the possible subtype constraints that may follow it, whatever their level[11]. Once developed the clause `COMPONENTS OF`, the type `Registration` defined above is therefore equivalent to:

```
Registration ::= SEQUENCE {
  surname        IA5String,
  first-name     IA5String,
  age            INTEGER,
  marital-status ENUMERATED {single, married, divorced,
                             widowed} }
```

The `COMPONENTS OF` clause is developped before checking the condition on distinct tags and applying the automatic tagging (but the choice of automatic-tag mode should be made before development because no tag of class context-specific appeared in the components, other than those of the clauses `COMPONENTS OF`).

It should be noted the difference between the type `Registration` above, which models quadruples, and the type below, which only has two components where the first is also structured (a 3-tuple):

```
OtherRegistration ::= SEQUENCE {
  description    Description,
  marital-status ENUMERATED {single, married, divorced,
                             widowed} }
```

By choosing one of the two forms, the specifier has to decide upon the meaning to be given to its specification, i.e. how it should be interpreted on the receiving side (since their encoding is not compatible, the choice should be made once and for all).

In Section 12.9 on page 244, we shall explain how the `SEQUENCE` type can be made extensible and add new components in a new version of

[11]It means that even if a constraint `WITH COMPONENTS` (see Section 13.9 on page 277) imposes that some components should be absent, this constraint will be ignored.

the specification[12]. In Section 13.9 on page 277, we will present the subtype constraint `WITH COMPONENTS` for constraining the components of a `SEQUENCE` type or only some of them.

12.2.2 Reference Manual

Type notation

SequenceType → `SEQUENCE` "{" "}"
  | `SEQUENCE` "{" *ExtensionAndException* *OptionalExtensionMarker* "}"
  | `SEQUENCE` "{" *ComponentTypeLists* "}"

⟨1⟩ The productions *ExtensionAndException* and *OptionalExtensionMarker* are defined on page 253.
⟨2⟩ This type has tag no. 16 of class `UNIVERSAL`, which is the same as that of type `SEQUENCE OF`.
⟨3⟩ This type can be constrained by a single value (production *SingleValue* on page 261), by type inclusion (production *ContainedSubtype* on page 263) and by constraints on its components (production *InnerTypeConstraints* on page 277).

ComponentTypeLists → *RootComponentTypeList*
  | *RootComponentTypeList* "," *ExtensionAndException* *ExtensionsAdditions* *OptionalExtensionMarker*
  | *RootComponentTypeList* "," *ExtensionAndException* *ExtensionsAdditions* *OptionalExtensionMarker* "," *RootComponentTypeList*
  | *ExtensionAndException* *ExtensionsAdditions* *OptionalExtensionMarker* "," *RootComponentTypeList*

⟨4⟩ For the third alternative, the *RootComponentTypeList* consists of two parts.

RootComponentTypeList → *ComponentTypeList*
ExtensionAdditions → "," *ExtensionAdditionList*
  | ϵ

ExtensionAdditionList → *ExtensionAddition* "," $\cdots^{+}$

ExtensionAddition → *ComponentType*
  | *ExtensionAdditionGroup*

[12] One may say that the clause `COMPONENTS OF` could simulate this extensibility in ASN.1:1990, but it would not offer so many advantages for making two versions of a protocol interwork.

ExtensionAdditionGroup → "`[[`" *ComponentTypeList* "`]]`"

ComponentTypeList → *ComponentType* "," $\cdots^{+}$

⟨5⟩ The identifiers in all the *ComponentType*s must be distinct.
⟨6⟩ In a module that includes the clause `AUTOMATIC TAGS` in its header, the root and the extensions of *ComponentTypeLists* are automatically tagged if they contain no type tagged by the specifier.
⟨7⟩ In a module that includes the clause `AUTOMATIC TAGS` in its header, if no type is tagged by the specifier in the root *RootComponentTypeList* (possibly made of two parts), none of the types of the *ExtensionAdditionList* can be tagged by the specifier. This rule prevents the belated introduction of a tag that would disturb the automatic tagging mode and would not preserve the upward compatibility.
⟨8⟩ The decision of automatic tagging of *ComponentTypeLists* should be taken *before* the expansion of the clauses `COMPONENTS OF` (see rule ⟨19⟩ on the following page), but the tagging is performed after this expansion (i.e. the automatic tagging is switched off if tags are actually present in *ComponentTypeLists*, but not if tags are present in the components brought in by a `COMPONENTS OF` clause). Consequently, if a `COMPONENTS OF` clause inserted components that were already tagged by the specifier, they are also tagged in `EXPLICIT` or `IMPLICIT` mode according to their respective type.
⟨9⟩ The automatic tagging consists in associating to every component a tag of class context-specific starting from 0 by one-increment. The extension root (possibly in two parts) is tagged first before the extensions.
⟨10⟩ The automatic tagging is always of mode `IMPLICIT`, except when a component is an untagged `CHOICE`, an untagged open type *ObjectClassFieldType* (see rule ⟨4⟩ on page 347), or an untagged parameter *DummyReference* that is a type (see rule ⟨17⟩ on page 386) because these are always tagged in `EXPLICIT` mode.
⟨11⟩ Even if some components of *SequenceType* are marked `ABSENT` in a subtype constraint `WITH COMPONENTS` (see Section 13.9 on page 277), they are taken into account by the automatic tagging algorithm.
⟨12⟩ Every *ExtensionAdditionGroup* delimits with double square brackets the component(s) added in a particular version of the module.

ComponentType → *NamedType*
| *NamedType* `OPTIONAL`
| *NamedType* `DEFAULT` <u>*Value*</u>
| `COMPONENTS OF` <u>*Type*</u>

⟨13⟩ In the `DEFAULT` clause, *Value* must be of the *Type* indicated in *NamedType* or of a type that is compatible with it according to the semantic model of ASN.1 (see Section 9.4 on page 121).
⟨14⟩ For every series of successive *ComponentType*s marked `OPTIONAL` or `DEFAULT` and appearing in *RootComponentTypeList* (possibly made of two parts) as well as in *ExtensionAdditionList*, the *Type*s must have tags that are distinct and different from the tag of the next mandatory *ComponentType* (i.e. not marked `OPTIONAL` or `DEFAULT`) that syntactically follows (if it exists).
⟨15⟩ The previous rule does not require the tags of *ExtensionAdditionList* to be canonically ordered as it is the case for the `SET` and `CHOICE` types (see rule ⟨15⟩ on page 228).
⟨16⟩ If the *SequenceType* is extensible or if one of its components is an extensible and untagged `CHOICE`, the rule ⟨16⟩ on page 255 describes a virtual transformation to be applied before checking the condition on distinct tags. This rule need not be applied when the module contains the `AUTOMATIC TAGS` in its header.
⟨17⟩ When *NamedType* is an untagged `CHOICE` type that includes no extension marker "...", the tags of all the alternatives of the `CHOICE` must be referred to when checking the condition on distinct tags.
⟨18⟩ In the `COMPONENTS OF` clause, *Type* must be a `SEQUENCE` type.
⟨19⟩ For every *ComponentType* that is actually a `COMPONENTS OF` clause, it must be considered to have been replaced (recursively in case of embedded `COMPONENTS OF`) in situ by the components of the referenced `SEQUENCE` type. This replacement should occur before checking all the other rules (see rules ⟨8⟩ and ⟨17⟩ above, in particular).
⟨20⟩ When the syntactic operator `COMPONENTS OF` is applied, if *Type* is a constrained `SEQUENCE` type, the subtype constraints that follow it are not taken into account and the components that are in the curly brackets of the `SEQUENCE` type are replicated word for word.
⟨21⟩ When the syntactic operator `COMPONENTS OF` is applied, if *Type* contains an extension marker, only the *RootComponentTypeList* (possibly made of two parts) is replicated.
⟨22⟩ When the syntactic operator `COMPONENTS OF` is applied, some inserted components may be of an implicitly extensible type if this type is imported from a module that contains the `EXTENSIBILITY IMPLIED` clause in its header. Similarly, the `EXPLICIT`/`IMPLICIT` tagging mode of every component's type is characteristic of the module where they are defined.

⟨23⟩ When the `COMPONENTS OF` clause appears in *ExtensionAdditionList*, each inserted component is considered as an *ExtensionAddition* (but not as an *ExtensionAdditionGroup*).

NamedType → identifier *Type*

⟨24⟩ If *Type* is a *SelectionType*, the rule ⟨2⟩ on page 240 justifies the two identifiers which appear in this case.

Value notation

SequenceValue → "{" *NamedValue* "," ···* "}"

⟨25⟩ There should be only one *NamedValue* for each *NamedType* not marked `OPTIONAL` nor `DEFAULT` in the *SequenceType*, and one or none for a *NamedType* marked thus.
⟨26⟩ The *NamedValue*s should appear in the same order as the *Named-Type*s in the type definition.
⟨27⟩ The notation "{}" should be used only if all the *ComponentType*s are marked `OPTIONAL` or `DEFAULT` in the type definition (or if the corresponding type is `SEQUENCE` {}, but this is of no need!).
⟨28⟩ The rules above also apply in presence of an extension marker "..." in the corresponding *SequenceType*.
⟨29⟩ If some components of an *ExtensionAdditionGroup* are present, all the mandatory components (i.e. not marked `OPTIONAL`, nor `DEFAULT`) in all the *ExtensionAdditionGroup*s which syntactically precede them should be present because ASN.1 only allows extensions one after the other.
⟨30⟩ The group of all *NamedValue*s pertaining to the same *Extension-AdditionGroup* is optional. But if a group is present then a *NamedValue* should be given for every mandatory *NamedType* of this *ExtensionAd-ditionGroup* (see also the preceding rule).

NamedValue → identifier *Value*

⟨31⟩ identifier is one of those appearing in the *SequenceType*.

12.3 The constructor `SET`

12.3.1 User's Guide

If the component order of the `SEQUENCE` type does not matter, the keyword `SET` is used for modeling such a non-ordered structure:

```
Description ::= SET { surname    IA5String,
                      first-name IA5String,
                      age        INTEGER }
```

In this case, the application can provide the components to the encoder in the best order for it[13]. Moreover, the encoder may also send these components in a different order as we shall see in Part III on the encoding rules.

The guide [ETSI114] quotes a textbook case of the `SET` type for the 'B-ISDN DSS2 UNI' protocol of the Integrated Services Digital Network (ISDN) where it is used for specifying that some information elements of the message may appear in any order. This would have been difficult, if not impossible, to express it in any other notation than ASN.1. However, it is recommended to use the `SEQUENCE` type whenever possible because it requires much less processing time for encoding and decoding the data (see Part III on page 391).

Like the `SEQUENCE` type, the `SET` type can use the `OPTIONAL` and `DEFAULT` markers. But the tagging rule is simpler: all the components (either mandatory or optional) of a `SET` should have distinct tags. The clause `COMPONENTS OF` can insert only components of another `SET` type. All the other rules of the `SEQUENCE` type apply to the `SET` type as well.

In Section 12.9 on page 244, we shall explain how a `SET` type can be made extensible and given new components in a subsequent version of the specification. In Section 13.9 on page 277, we will present the subtype constraint `WITH COMPONENTS`, which allows constraining every component of a `SET` type.

[13]This concept of unordered data was initially introduced in the [X.409] standard: e-mail applications may have to handle data of great size that could be sent in any order.

12.3.2 Reference Manual

Type notation

SetType → `SET` "{" "}"
  | `SET` "{" *ExtensionAndException* *OptionalExtensionMarker* "}"
  | `SET` "{" *ComponentTypeLists* "}"

⟨1⟩ The productions *ExtensionAndException* and *OptionalExtension-Marker* are defined on page 253.
⟨2⟩ This type has tag no. 17 of class `UNIVERSAL`, which is the same as that of type `SET OF`.
⟨3⟩ This type can be constrained by a single value (production *Single-Value* on page 261), by type inclusion (production *ContainedSubtype* on page 263) or by constraints on its components (production *InnerType-Constraints* on page 277).

ComponentTypeLists → *RootComponentTypeList*
  | *RootComponentTypeList* "," *ExtensionAndException*
  ExtensionsAdditions *OptionalExtensionMarker*
  | *RootComponentTypeList* "," *ExtensionAndException*
  ExtensionsAdditions *OptionalExtensionMarker* ","
  RootComponentTypeList
  | *ExtensionAndException* *ExtensionsAdditions*
  OptionalExtensionMarker "," *RootComponentTypeList*

⟨4⟩ For the third alternative, the *RootComponentTypeList* is made of two parts.

RootComponentTypeList → *ComponentTypeList*
ExtensionAdditions → "," *ExtensionAdditionList*
  | ϵ
ExtensionAdditionList → *ExtensionAddition* "," $\cdots^{+}$
ExtensionAddition → *ComponentType*
  | *ExtensionAdditionGroup*
ExtensionAdditionGroup → "`[[`" *ComponentTypeList* "`]]`"
ComponentTypeList → *ComponentType* "," $\cdots^{+}$

⟨5⟩ The identifiers in all the *ComponentType*s must be distinct.
⟨6⟩ In a module that includes the clause `AUTOMATIC TAGS` in its header, the root and the extensions of *ComponentTypeLists* are automatically tagged if they contain no type tagged by the specifier.
⟨7⟩ In a module that includes the clause `AUTOMATIC TAGS` in its header,

if no component of the *RootComponentTypeList* (possibly made of two parts) is tagged by the specifier, then none of the types of the *ExtensionAdditionList* can be tagged by the specifier. This rule prevents the belated introduction of a tag that would disturb the automatic tagging mode and would not preserve the upward compatibility.
⟨8⟩ The decision of tagging automatically *ComponentTypeLists* should be taken before the expansion of the clauses `COMPONENTS OF` (see rule ⟨19⟩ on page 224), but the tagging is performed after this expansion (i.e. the automatic tagging is switched off if tags are actually present in *ComponentTypeLists*, but not if tags are present in the components brought in by the `COMPONENTS OF` clause). Consequently, if the `COMPONENTS OF` clause has inserted components that were already tagged by the specifier, they are also tagged in `EXPLICIT` or `IMPLICIT` mode according to their type.
⟨9⟩ The automatic tagging consists in associating to every component a tag of class context-specific starting from 0 by one-increment. The extension root (possibly in two parts) is tagged first before the extensions.
⟨10⟩ The automatic tagging is always of mode `IMPLICIT`, except when the component is an untagged `CHOICE`, an untagged open type *ObjectClassFieldType* (see rule ⟨4⟩ on page 347), or an untagged parameter *DummyReference* that is a type (see rule ⟨17⟩ on page 386) because these are always tagged in `EXPLICIT` mode.
⟨11⟩ Even if some components of *SetType* are marked `ABSENT` in a subtype constraint `WITH COMPONENTS` (see Section 13.9 on page 277), they are taken into account by the automatic tagging algorithm.
⟨12⟩ Every *ExtensionAdditionGroup* delimits with double square brackets the component(s) added in a particular version of the module.

$$
\begin{aligned}
\textit{ComponentType} \rightarrow\ & \textit{NamedType} \\
\mid\ & \textit{NamedType}\ \texttt{OPTIONAL} \\
\mid\ & \textit{NamedType}\ \texttt{DEFAULT}\ \underline{\textit{Value}} \\
\mid\ & \texttt{COMPONENTS OF}\ \underline{\textit{Type}}
\end{aligned}
$$

⟨13⟩ In the `DEFAULT` clause, *Value* must be of the *Type* indicated in *NamedType* or of a type that is compatible with it according to the semantic model of ASN.1 (see Section 9.4 on page 121).
⟨14⟩ The *ComponentTypes* must have distinct tags whether they appear in the root or in the extensions.
⟨15⟩ In *ExtensionAdditionList*, the tags of each component must be canonically ordered as follows:

a) first, the tags of class `UNIVERSAL`, followed by the tags of class

APPLICATION, then the tags of context-specific class, and finally the tags of class PRIVATE;

b) for each class, the numbers must appear in increasing order.

⟨16⟩ If the *SetType* is extensible or if one of its components is an extensible and untagged CHOICE, the rule ⟨16⟩ on page 255 describes a virtual transformation to be applied for checking the condition on distinct tags. This rule needs not be applied when the module contains the AUTOMATIC TAGS in its header.

⟨17⟩ When the *NamedType* is an untagged CHOICE type that includes no extension marker "...", the tags of all the alternatives of this CHOICE must be referred to when checking the condition on distinct tags.

⟨18⟩ In the COMPONENTS OF clause, *Type* must be of type SET.

⟨19⟩ For every *ComponentType* that is actually a COMPONENTS OF clause, it must be considered to have been replaced (recursively in case of embedded COMPONENTS OF) in situ by the components of the referenced SET type. This replacement should occur before checking all the other rules (see rules ⟨8⟩ and ⟨17⟩ above in particular).

⟨20⟩ When the syntactic operator COMPONENTS OF is applied, if *Type* is a constrained SET type, the subtype constraints that follow it are not taken into account and the components that are in the curly brackets of the SET type are replicated word for word.

⟨21⟩ When the syntactic operator COMPONENTS OF is applied, if *Type* contains an extension marker, only the *RootComponentTypeList* (possibly made of two parts) is replicated.

⟨22⟩ When the syntactic operator COMPONENTS OF is applied, some inserted components may be of an implicitly extensible type if this type is imported from a module that contains the EXTENSIBILITY IMPLIED clause in its header. Similarly, the EXPLICIT/IMPLICIT tagging mode of every component's type is characteristic of the module where they are defined.

⟨23⟩ When the COMPONENTS OF clause appears in *ExtensionAdditionList*, each inserted component is considered as an *ExtensionAddition* (but not as an *ExtensionAdditionGroup*).

NamedType → identifier $\underline{\textit{Type}}$

⟨24⟩ If *Type* is a *SelectionType*, the rule ⟨2⟩ on page 240 justifies the two identifiers that appear in this case.

Value notation

SetValue → "{" *NamedValue* "," $\cdots^*$ "}"

⟨25⟩ There must be only one *NamedValue* for each *NamedType* not marked `OPTIONAL` nor `DEFAULT` in the *SetType*, and one or none for a *NamedType* marked thus.
⟨26⟩ The *NamedValue*s need not appear in the same order as the *NamedType*s in the type definition.
⟨27⟩ The “{}” notation must be used only if all the *ComponentType*s are marked `OPTIONAL` or `DEFAULT` in the type definition (or if the corresponding type is `SET {}`, which is devoid of interest).
⟨28⟩ The rules above also apply in presence of an extension marker “. . .” in the corresponding *SetType*.
⟨29⟩ If some components of an *ExtensionAdditionGroup* are present, all the mandatory components (i.e. not marked `OPTIONAL` nor `DEFAULT`) in all the *ExtensionAdditionGroup*s that syntactically precede them must be present because ASN.1 only allows extensions to be given one after the other.
⟨30⟩ The group of all *NamedValue*s pertaining to the same *ExtensionAdditionGroup* is optional. But if a group is present, then a *NamedValue* must be indicated for each mandatory *NamedType* of this *ExtensionAdditionGroup* (see also the preceding rule).

NamedValue → identifier <u>*Value*</u>

⟨31⟩ identifier is one of those appearing in the *SetType*.

12.4 The constructor `SEQUENCE OF`

12.4.1 User’s Guide

The `SEQUENCE OF` type is equivalent to the dynamic arrays or lists of some programming languages: all the elements are of the same type but the number of the elements is not necessarily known beforehand.

It can, for example, model a directory preserving the entry order:

```
Directory ::= SEQUENCE OF DirectoryEntry
```

or the result of a horse race for which the arriving order information should be present:

```
PariTierce DEFINITIONS ::=
BEGIN
SteepleChase ::= SEQUENCE OF INTEGER
END
```

A winning combination would then be written (in curly brackets for sure!):

```
winningCombination SteepleChase ::= {5, 2, 12}
no-one-arrived SteepleChase ::= {}
```

But in principle, it obviously remains possible to have the same value several times in a list. In that case, the application designers should decide on the semantics to give to the fact that a value occurs several times.

In order to make specification-related comments and documents easier to write, ASN.1 provides a meta-notation[14] beginning with the at-sign "`@`", followed by the module name, the name of the `SEQUENCE OF` type, and finally an element number in the value or the "`*`" symbol for denoting all the elements. It can be used to add to the type `SteepleChase` informal constraints that cannot be specified in ASN.1, such as:

```
-- @PariTierce.SteepleChase.* must be distinct
-- @PariTierce.SteepleChase.0 = 5 (horse no. 5 always wins!)
```

It is a proposition for a notation and no specifier is bound to use it; it has the undeniable advantage of providing more homogeneous formal comments. If they appear within the ASN.1 module, the name of the module and the at-sign "`@`" can be omitted:

```
-- SteepleChase.* must be distinct
-- SteepleChase.0 = 5 (horse no. 5 always wins!)
```

In Section 13.5 on page 266, we present the `SIZE` constraint that limits the number of elements of a value of type `SEQUENCE OF`, and in Section 13.8 on page 275, the `WITH COMPONENT` constraint that enables applying a constraint to each element of a list.

Historically the `SEQUENCE OF` type has the same tag of class `UNIVERSAL` as the `SEQUENCE` type, this has unfortunate consequences on the length of the BER encoding (see Chapter 18). [Lar96, Chapter 7] quotes the case of the [X.400] message handling system where this peculiarity enabled updating the 1984 version to the 1988 version while ensuring the encoding compatibility.

[14]The prefix 'meta-' denotes that this notation cannot be used in the formal part of an ASN.1 module. It should therefore be used within comments, in a specification documentation, in a book... It should not be confused with the 'pointed' relational constraints that are introduced in Section 15.7 on page 341 that deals with information object classes.

12.4.2 Reference Manual

Type notation

SequenceOfType → `SEQUENCE OF` *Type*

⟨1⟩ This type has tag no. 16 of class `UNIVERSAL`, which is the same as that of type `SEQUENCE`.
⟨2⟩ This type can be constrained by a single value (production *SingleValue* on page 261), by type inclusion (production *ContainedSubtype* on page 263), by size (production *SizeConstraint* on page 267) and by constraint on its elements (production *InnerTypeConstraints* on page 277).

TypeWithConstraint → `SEQUENCE` *Constraint* `OF` *Type*
| `SEQUENCE` *SizeConstraint* `OF` *Type*
| `SET` *Constraint* `OF` *Type*
| `SET` *SizeConstraint* `OF` *Type*

⟨3⟩ *Constraint* should be a subtype constraint (or a combination of constraints) by a single value (of the form "`SEQUENCE (v) OF T`", see rule ⟨2⟩ on page 261), by type inclusion (of the form "`SEQUENCE (INCLUDES T1) OF T`", see rule ⟨4⟩ on page 263), by size (of the form "`SEQUENCE SIZE (n) OF T`", see below) or a constraint on the elements (of the form "`SEQUENCE (WITH COMPONENT (`*`Constraint`*`)) OF T`", see rule ⟨1⟩ on page 277). Still, it is recommended to use only a `SIZE` constraint in this place for clarity's sake, but also because other constraints may not be supported by tools.
⟨4⟩ Only the size constraint (keyword `SIZE`) was allowed between the keyword `SEQUENCE` and `OF` in ASN.1:1990.

SizeConstraint → `SIZE` *Constraint*

⟨5⟩ Although the production *Constraint* (see on page 293) can be derived in "`(SIZE` *`Constraint`*`)`" (*with* round brackets), the production *SizeConstraint* (*without* round brackets) is maintained to ensure the compatibility with ASN.1:1990.
⟨6⟩ In *SizeConstraint*, *Constraint* must be a subtype constraint valid for the parent type `INTEGER(0..MAX)`.
⟨7⟩ The unit of `SIZE` is the element.

Value notation

SequenceOfValue → "{" *Value* "," ···* "}"

⟨8⟩ Every *Value* must be of the *Type* appearing in the associated `SEQUENCE OF` type or of a type that is compatible with *Type* according to the semantic model of ASN.1 (see Section 9.4 on page 121).
⟨9⟩ The same value can appear several times in the *SequenceOfValue* (in this case, all the occurrences are transmitted). In that case, the application designer should decide on the semantics to be given by the application if a value occurs more than once in a *SequenceOfValue*.

12.5 The constructor `SET OF`

12.5.1 User's Guide

If the element ordering is not required during the data transfer, the keyword `SET OF` can be used. It stands for the mathematical notion of bag: a non-ordered set including elements of the same type, where some of them may appear several times.

Having already modeled the pari tiercé using the `SEQUENCE OF` type, the `SET OF` type can now be used for representing the National lottery for instance:

```
LotteryDraw ::= SET OF INTEGER
  -- LotteryDraw.* must be distinct
  -- example:  draw LotteryDraw ::= {17, 45, 5, 25, 12, 38}
```

It is recommended to use the `SEQUENCE OF` type whenever possible: indeed, when using canonical encoding rules such as the CER or the canonical PER (see Part III on page 391), the `SET OF` type requires dynamically sorting the encoding of every value before transmitting it.

The contraints `SIZE` (see Section 13.5 on page 266) and `WITH COMPONENT` (see Section 13.8 on page 275) previously defined for the `SEQUENCE OF` type also apply to the `SET OF` type.

12.5.2 Reference Manual

Type notation

SetOfType → `SET OF` *Type*

⟨1⟩ This type has tag no. 17 of class `UNIVERSAL`, which is the same as that of type `SET`.
⟨2⟩ This type can be constrained by a single value (production *Single-Value* on page 261), by type inclusion (production *ContainedSubtype* on

page 263), by size (production *SizeConstraint* on page 267) and by constraint on its elements (production *InnerTypeConstraints* on page 277).

TypeWithConstraint → `SEQUENCE` *Constraint* `OF` *Type*
| `SEQUENCE` *SizeConstraint* `OF` *Type*
| `SET` *Constraint* `OF` *Type*
| `SET` *SizeConstraint* `OF` *Type*

⟨3⟩ *Constraint* must be a subtype constraint (or a combination of constraints) by a single value (of the form "`SET (v) OF T`", see rule ⟨2⟩ on page 261), by type inclusion (of the form "`SET (INCLUDES T1) OF T`", see rule ⟨4⟩ on page 263), by size (of the form "`SET SIZE (n) OF T`", see below) or a constraint on the elements (of the form "`SET (WITH COMPONENT (`*`Constraint`*`)) OF T`", see rule ⟨1⟩ on page 277). Still, it is recommended to use only a `SIZE` constraint in this place for clarity's sake, but also because other constraints may not be supported by ASN.1 tools.
⟨4⟩ Only the size constraint (keyword `SIZE`) was allowed between the keyword `SET` and `OF` in ASN.1:1990.

SizeConstraint → `SIZE` *Constraint*

⟨5⟩ Although the production *Constraint* (see on page 293) can be derived in "`(SIZE `*`Constraint`*`)`" (*with* round brackets), the production *SizeConstraint* (*without* round brackets) is maintained to ensure the compatibility with ASN.1:1990.
⟨6⟩ In *SizeConstraint*, *Constraint* must be a subtype constraint valid for the parent type `INTEGER(0..MAX)`.
⟨7⟩ The unit of `SIZE` is the element.

Value notation

SetOfValue → "{" *Value* "," ⋯* "}"

⟨8⟩ Every *Value* must be of the *Type* that appears in the corresponding `SET OF` type or of a type that is compatible with *Type* according to the semantic model of ASN.1 (see Section 9.4 on page 121).
⟨9⟩ The same value can appear several times in the *SetOfValue* (in this case, all the occurrences are transmitted). In that case, the application designer should decide on the semantics to be given by the application if a value occurs more than once in a *SetOfValue*.

12.6 The constructor `CHOICE`

12.6.1 User's Guide

The constructor `CHOICE` gives the choice (also called 'union') between several alternatives:

```
Afters ::= CHOICE { cheese  [0] IA5String,
                    dessert [1] IA5String }
```

An alternative is denoted by an identifier, which is a word beginning with a lower-case letter. This identifier is not encoded and it makes it possible to build unambiguous abstract values[15]; for this reason, the identifiers of a `CHOICE` type must be distinct. Their names should be as self-explicit as possible so that the role of each alternative could be easily understood.

A value of type `CHOICE` features the identifier of the chosen alternative followed by the symbol "`:`"[16], and a value complying with the type of this alternative:

```
mine Afters ::= dessert:"profiteroles"
```

We should point out a detail seldom noticed by beginners when specifying. The `CHOICE` type models two pieces of information: the chosen alternative (the identifier) and the value associated with this alternative. Therefore, in the previous value `mine`, we first indicate that the meal ends with a `dessert`, and also that this dessert will be `"profiteroles"`. There is no need of defining an enumerated type before the `CHOICE` type since the `CHOICE` type enumerates the alternatives in its 'first column':

```
Alternative ::= ENUMERATED {cheese, dessert}
Afters ::= CHOICE { cheese  [0] IA5String,
                    dessert [1] IA5String }
```

A BER decoder will rely on the received tag to determine the alternative that has been chosen and decode the value according to this

[15]These identifiers have been mandatory since 1994.

[16]This symbol was introduced in 1992 in a technical corrigendum. Without this corrigendum, a full-ASN.1 parsing would have been impossible (see [Ste93, page 116] or [Rin95] for some examples of these ambiguous definitions) because some of these ambiguities could not have been removed even when reaching the end of the module. This technical corrigendum certainly did not go as far as ensuring an upward compatibility, but it still remained quite easy to insert the "`:`" symbol in values of type `CHOICE` to update the obsolete specifications.

alternative. The tags of the CHOICE alternatives must therefore be distinct. The CHOICE constructor is different from the SEQUENCE and SET constructor in the sense that it has no (default) UNIVERSAL class tag. Indeed, the CHOICE type does not exist 'as is'; it is only a collection of several types among which one of them is chosen to be encoded with its associated tag.

A CHOICE type can potentially adopt the tag of any of its alternatives and we should bear this specificity in mind in case this type is embedded in a SEQUENCE or SET type, or in another CHOICE type. Thus the type U that follows is not semantically correct because it includes the tag [0] twice (alternative a and component x) and the tag [1] twice (alternatives b and c):

```
T ::= CHOICE { a [0] INTEGER,
               b [1] NULL }
U ::= SET { x [0] REAL,
            y T,
            z CHOICE { c [1] BIT STRING,
                       d [2] OCTET STRING }}
```

If the specifier has not inserted the AUTOMATIC TAGS clause in the module header, it is recommended to add a tag of class context-specific before every CHOICE type:

```
U ::= SET { x [0] REAL,
            y [1] T,
            z [2] CHOICE { c [1] BIT STRING,
                           d [2] OCTET STRING }}
```

We very often come across the CHOICE type in ASN.1 specifications since it is almost always the (highest level) type of the PDU transmitted by the application: it generally collects 'questions' and 'answers' that should be exchanged. The following example gives a simplified version of the PDU of the Remote Operation Service Element (ROSE) [ISO13712-1]:

```
ROS ::= CHOICE {
  invoke       [1] Invoke,
  returnResult [2] ReturnResult,
  returnError  [3] ReturnError,
  reject       [4] Reject }
```

For another example, the reader may refer to the (extensible) PDU defined in the case study on page 39.

The `CHOICE` type is also used when some information can be transmitted in several different ways (for example, different character string types for the same data) as in the [X.520] standard:

```
CHOICE { teletexString   TeletexString,
         printableString PrintableString,
         universalString UniversalString }
```

In some cases, however, a parameterization of the type should be preferred to using a `CHOICE` type. This possibility is described in Chapter 17.

As we shall see in Section 12.9 on page 244, if it is estimated that extra types may be added to the specification in the future then, in the first version of this specification, we should consider using an extensible `CHOICE` type with only one alternative.

In Section 13.9 on page 277, we will present the `WITH COMPONENTS` subtype constraint, which enables restricting the number of alternatives of a `CHOICE` and constraining every one of its alternative type.

12.6.2 Reference Manual

Type notation

ChoiceType → `CHOICE` "{" *AlternativeTypeLists* "}"

⟨1⟩ This type has no tag of class `UNIVERSAL`. It can have the tag of any of its alternatives.
⟨2⟩ This type can be constrained by a single value (production *SingleValue* on page 261), by type inclusion (production *ContainedSubtype* on page 263) and by constraints on its alternatives (production *InnerTypeConstraints* on page 277).

AlternativeTypeLists → *RootAlternativeTypeList*
  | *RootAlternativeTypeList* "," *ExtensionAndException*
  ExtensionAdditionAlternatives *OptionalExtensionMarker*

⟨3⟩ A `CHOICE` type definition can include two extension makers "...", but contrary to the `SEQUENCE` and `SET` types (see rules ⟨4⟩ on page 222 and ⟨4⟩ on page 227), its extension root is made of a single part. The second extension marker is only meant to emphasize the insertion point where new extensions can be added.

RootAlternativeTypeList → *AlternativeTypeList*
ExtensionAdditionAlternatives →
  "," *ExtensionAdditionAlternativesList*
  | ϵ

ExtensionAdditionAlternativesList →
  ExtensionAdditionAlternative
  | *ExtensionAdditionAlternativesList* “,” *ExtensionAdditionAlternative*

ExtensionAdditionAlternative →
  ExtensionAdditionGroupAlternatives
  | *NamedType*

ExtensionAdditionGroupAlternatives →
  “`[[`” *AlternativeTypeList* “`]]`”

AlternativeTypeList → *NamedType* “,” $\cdots^{+}$

⟨4⟩ In a module that includes the `AUTOMATIC TAGS` clause in its header, the root and the extensions of the *AlternativeTypeLists* are automatically tagged if no root alternative has been tagged by the specifier.
⟨5⟩ In a module that includes the `AUTOMATIC TAGS` clause in its header, if no type of the *RootAlternativeTypeList* is tagged by the specifier, none of the types of the *ExtensionAdditionAlternativesList* can be tagged by the specifier. This rule prevents the belated introduction of a tag that would disturb the automatic tagging mode and would not preserve the upward compatibility.
⟨6⟩ The automatic tagging consists in associating to every alternative a tag of context-specific class starting from 0 by one-increment. The extension root is tagged first before the extensions.
⟨7⟩ The automatic tagging is always of mode `IMPLICIT`, except when an alternative is an untagged `CHOICE` type, an untagged open type *ObjectClassFieldType* (which replaces the `ANY` type since 1994, see rule ⟨4⟩ on page 347) or an untagged parameter *DummyReference* that is a type (see rule ⟨17⟩ on page 386) because these are always tagged in `EXPLICIT` mode.
⟨8⟩ Even if some alternatives of *ChoiceType* are marked `ABSENT` in a subtype constraint `WITH COMPONENTS` (see Section 13.9 on page 277), they are taken into account by the automatic tagging algorithm.
⟨9⟩ The tags of all the alternatives must be distinct. The list consists of the tags of all the alternatives of the *ChoiceType*, except when the alternative is a *ChoiceType* itself, not preceded by a tag and with no extension marker (see rule ⟨1⟩ on the preceding page), in which case the list of tags of the inner *ChoiceType* is appended to the former list, and so on. If one of the alternatives is an extensible `CHOICE` type, the rule ⟨16⟩ on page 255 describes a virtual transformation to be applied before checking the condition on distinct tags (this rule needs not be applied if the module contains the `AUTOMATIC TAGS` clause in its header).

⟨10⟩ In the *ExtensionAdditionAlternativesList*, the tags must be canonically ordered according to the rule ⟨15⟩ on page 228.

NamedType → identifier <u>*Type*</u>

⟨11⟩ The identifiers in all the *NamedTypes* must be distinct.
⟨12⟩ If *Type* is a *SelectionType*, rule ⟨2⟩ on the next page justifies the two identifiers that appear in this case.
⟨13⟩ Every *ExtensionAdditionGroupAlternatives* delimits with double square brackets the alternative(s) added in a particular version of the module. The presence (or absence) of these double square brackets does not affect the encoding.

Value notation

ChoiceValue → identifier ":" <u>*Value*</u>

⟨14⟩ The symbol ":" was introduced by a technical corrigendum in 1993. This document is retroactive to all existing specifications. Without this symbol, an ASN.1 module could not be parsed properly.
⟨15⟩ *Value* must be of the type associated with the identifier in the corresponding *ChoiceType* or of a type that is compatible with this type according to the semantic model of ASN.1 (see Section 9.4 on page 121).
⟨16⟩ The presence of an extension marker "..." in the corresponding *ChoiceType* does not affect the previous rule.

12.7 Selecting a `CHOICE` alternative

12.7.1 User's Guide

Sometimes one wants to re-use the type of a `CHOICE` alternative while allowing for an easier specification updating (a single modification of the selected type implicitly spreads the changes over the whole specification), and keeping the semantic link to the `CHOICE` type (thereby bestowing the adequate meaning to the `CHOICE` type for reading purpose). The chosen type can be selected by the left angle bracket "<" preceded by the alternative to be extracted and followed by a reference to the `CHOICE` type as in:

```
Element ::= CHOICE { atomic-no INTEGER (1..103),
                     symbol    PrintableString }
MendeleievTable ::=
  SEQUENCE SIZE (103) OF symbol < Element
einsteinium symbol < Element ::= "Es"
```

The symbol "<" is called 'selection' in the ASN.1 standard. It is a syntactic operator (like `COMPONENTS OF`) that copies the type which appears in the corresponding alternative of the `CHOICE` type (leaving aside any subtype constraint applied to the `CHOICE` type). It is hardly ever used in practice.

12.7.2 Reference Manual

Type notation

SelectionType → identifier "<" *Type*

⟨1⟩ *Type* must be a reference to a `CHOICE` type.
⟨2⟩ identifier must be an alternative of the referenced `CHOICE` type. If *SelectionType* is the type (*NamedType*) of a component of a `SEQUENCE` or `SET` or the type of an alternative of a `CHOICE`, the identifier of *NamedType* is used to name the component and the identifier of *SelectionType* is used for selecting the alternative. If *SelectionType* is used elsewhere, (for example, directly behind the "`::=`" symbol), its identifier is only meant to select the alternative.
⟨3⟩ The operator "<" is syntactic: it consists in copying literally the type of the selected alternative of the `CHOICE` discarding any subtype constraint `WITH COMPONENTS` applied to this `CHOICE` type, i.e. all the alternatives that are syntactically in curly brackets after the keyword `CHOICE` are taken into account.
⟨4⟩ If a *SelectionType* is literally followed by a subtype constraint (i.e. "identifier < *Type*`(`*ConstraintSpec*`)`"), it is the `CHOICE` type that is constrained, which implies that this constraint is meaningless for the selection (see previous rule). If a reference to a *SelectionType* is literally followed by a subtype constraint, it is the selected type that is constrained, and not the `CHOICE` type.

Value notation

⟨5⟩ A value for *SelectionType* conforms to the production *Value* on page 109, and must be of the type of the alternative selected in the `CHOICE` type or of a type that is compatible with this type according to the semantic model of ASN.1 (see Section 9.4 on page 121).

12.8 The special case of the ANY type

This type has disappeared from the standard since 1994 and its use is now strongly inadvisable. We shall see at the end of this section how ANY types have been replaced since then.

12.8.1 User's Guide

The reason why we choose to introduce the 'any type' after the CHOICE type is for defining it conceptually as a CHOICE (with an infinite number of alternatives) generalized to all the types that can be defined with ASN.1. But rather than referencing an alternative by its identifier (a word beginning with a lower-case letter), we use the *type* of the alternative to indicate which is retained (a word beginning with an upper-case letter). But for this detail, a value of type ANY is denoted as a value of type CHOICE[17]:

```
v ANY ::= INTEGER:12
T ::= SEQUENCE { a BOOLEAN,
                 b REAL }
w ANY ::= T:{ a TRUE,
              b {314, 10, -2} }
```

As the CHOICE type, the ANY type has no (default) tag of class UNIVERSAL: in case of BER encoding, the tag depends on the type of the value to be transmitted.

The [X.409] standard recommended to "*use an Any to model a variable whose type is unspecified, or specified in another Recommendation*". As a consequence, the ANY type enabled the specifier to leave a gap in a specification when the exact type of some data was not known yet. It was originally meant to be used during the specification design phase provided that the sender and the receiver had agreed on a few types they could exchange. It would have been replaced later by a 'properly' defined type in a future version of the specification.

It was the case for the first edition of the Association Control Service Element standard (ACSE, [ISO8650-1]) for lack of agreement on the application entity format. In ACSE, the AE-title type had been first

[17]In 1993 the same technical corrigendum introduced the symbol ":" in CHOICE values (see on page 235) and in ANY values. It also forbade the keyword SEQUENCE (SET respectively) not followed by curly brackets, which was a shorthand notation for SEQUENCE OF ANY (SET OF ANY respectively).

specified as an `ANY` type and then replaced by an object identifier [Lar96, Chapter 9].

The unrestricted use of the `ANY` type was strongly deprecated. Besides, `ANY` was very often the component type of a `SEQUENCE` or a `SET` conditioned by the value of some other component of the same `SEQUENCE` or `SET`. This happens when the parameter type of a 'parameterized' error message depends on the error code. The clause `DEFINED BY` can indicate this semantic link between two components of the same `SEQUENCE` (or `SET`) type:

```
Error ::= SEQUENCE { code      INTEGER,
                     parameter ANY DEFINED BY code }
```

The component referenced after the `DEFINED BY` clause could be of type `INTEGER`, `OBJECT IDENTIFIER` or `CHOICE` between these two types[18]. Unfortunately ASN.1 provided no way of formally indicating what the possible values of this semantic link could be. This lack of formalization prevented it from being treated automatically by compilers so that application designers sometimes had to write 'manually' part of the software dealing with these references to implement the whole specification.

In fact, the standard recommended defining an association list by means of comments within the module. For the type `Error` above, one could have written:

```
--   code |   parameter type
-- ======|=====================
--   0    | NULL
--   1    | INTEGER
--   2    | SEQUENCE OF INTEGER
--   3    | SEQUENCE { param1 T1, param2 T2 }
```

In some specifications, the association table would have been modeled thanks to appropriate macro instances. Unfortunately ASN.1 macros were as badly supported as the `DEFINED BY` clause, as we shall see in Chapter 16. In the Remote Operation Service Element standard (ROSE, [ISO9072-2]), we find the following comment where it is explained that the pairs ⟨`operation-value`, `argument`⟩ correspond to the

[18]We have already mentioned in the history on page 63 that this was the only difference between the CCITT X.208:1988 recommendation and the ISO 8824:1990 standard.

registration modeled by the `OPERATION` macro instances (see Section 16.5 on page 371):

```
ROIVapdu ::= SEQUENCE {
  invokeID        InvokeIDType,
  linkedID        [0] IMPLICIT InvokeIDType OPTIONAL,
  operation-value OPERATION,
  argument        ANY DEFINED BY operation-value OPTIONAL
      -- ANY is filled by the single ASN.1 data
      -- type following the keyword ARGUMENT in the type
      -- definition of a particular operation -- }
```

If the component referenced after the clause `DEFINED BY` were of type `OBJECT IDENTIFIER` then the table was dynamically extensible since the types needed only to be registered in the universal registration tree (see Section 10.8 on page 153).

For the X.500 directory service, a database can be queried using test values and a comparison operator for some attributes. Every attribute has an object identifier in the registration tree so that the value type is conditioned by this object identifier[19]:

```
AttributeValueAssertion ::=
  SEQUENCE { attribute OBJECT IDENTIFIER,
             is        Operator,
             value     [0] ANY DEFINED BY attribute }
Operator ::= ENUMERATED { equalTo(0), greaterThan(1),
  greaterOrEqualTo(2), lessThan(3), lessOrEqualTo(4),
  notEqualTo(5) }
```

In the [X.509] standard, the nature of the parameters of the authentication procedure depends on the algorithm:

```
AlgorithmIdentifier ::= SEQUENCE {
  algorithm  OBJECT IDENTIFIER,
  parameters ANY DEFINED BY algorithm OPTIONAL }
```

But the `ANY` type and its `DEFINED BY` clause had serious drawbacks: the association table could not be formalized (and the notation of macro instances was of no help for the application designer); the component referenced in the `DEFINED BY` clause had to belong to the same `SEQUENCE` or

[19]The version given here is adapted from the types `AttributeAssertion` and `AttributeValueAssertion` of the [X.501] standard that we shall discuss further in Chapter 15 when describing how the link `DEFINED BY` can be defined formally using the information object classes.

`SET`; some ambiguous subtyping of `ANY` types had been detected [Ste93]. All these problems were solved in 1994 by the introduction of the concepts of information object class (and particularly the `TYPE-IDENTIFIER` class, see Section 15.9 on page 355), open type and tabular constraint (see Chapter 15), together with parameterization (see Chapter 17).

12.8.2 Reference Manual

The type `ANY` and its value notation are described in [ISO8824-1, annex F]. We do not present them here since their use is strongly unrecommended and is not standardized any more since 1994.

12.9 Type extensibility

12.9.1 User's Guide

We have already breached the problem of extensibility in the case of the `ENUMERATED` type in Section 10.4 on page 135. We now go into more detail on its principle and present its main advantages before describing how the constructed types `SEQUENCE`, `SET` or `CHOICE` can be made extensible. We shall come back on this subject in Section 13.12 on page 291 to apply the concept of extensibility to subtype constraints.

Long before the notion of extensibility was explicitly introduced in the ASN.1:1994 standard, many protocol specifications included a section called "*extensibility rules*" to describe the behavior that a compiler was to adopt when receiving values outside the set of values of a type.

Indeed, the basic encoding rules (see Chapter 18) had been conceived right from the start to implicitly take into account this notion using tags: a BER decoder ignores any tag that does not correspond to what is expected [ISO8823-1, clause 8.5.1.a]. Thus, in the Common Management Information Protocol (CMIP, [ISO9596-1]), we could read:

> *7.5.1.1 When processing incoming CMIP-A-ASSOCIATE-Information, the accepting CMIPM shall:*
>
> - *ignore all tagged values that are not defined in the abstract syntax of this Recommendation, and*
> - *ignore all unknown bit name assignments within a* `BIT STRING`*.*

7.5.1.2 The abstract syntax name may be used when the presentation data values are modified to include:

- *new system management operations,*
- *new tagged elements within a `SET` or `SEQUENCE`,*
- *new bit name assignments within a `BIT STRING`,*
- *new named numbers for an `INTEGER`, and*
- *new named enumerations within an `ENUMERATED`.*

In so open a world as OSI is meant to be, it seems difficult to impose that applications should communicate with (exactly) the same version of abstract syntax. As the environment of an application is inclined to regularly changes, a protocol is very often used simultaneously in different versions. An application designer is therefore free to adopt the new version of a protocol whenever it is judged fit.

But the only way of ensuring interworking (i.e. demanding that a decoder, for example, that uses an old version of the protocol could discard data specific to a new version), is to adapt the encoding rules to provide the appropriate manifold for these extensions so that the decoder could detect them without knowing how to interpret them for all that. The dual problem should also be tackled: a decoder generated from a new version of the abstract syntax, receiving data that conform to an old version should recover from not receiving the expected extensions (see Figure 12.1 on page 247).

As a result, the encoders and decoders generated from the very first version of the abstract syntax should include the appropriate support[20] for the extension management since a version 1 encoder can send data to a version 2 decoder while indicating that the data are not extended, or a version 1 decoder can receive data from a version 2 encoder and should be able to ignore extensions. The capacity of dealing with extensible types is clearly one of ASN.1 strong points compared to other abstract syntax notations like those introduced in Chapter 24.

This analysis induces us to require the specifier to indicate, from the very first version of a protocol, the data types that could be extended in the future. Note that the point is to keep the type structure since, otherwise the specifications would not be upward compatible (replacing a type with another generally change the transfer syntax).

[20]This support appearing in the abstract syntax makes no assumption as for the encoding rules that are going to be used.

To indicate that a type is extensible, we insert an extension marker "..." in its definition. In ASN.1, the only extensible types are `ENUMERATED`, `SEQUENCE`, `SET` and `CHOICE`:

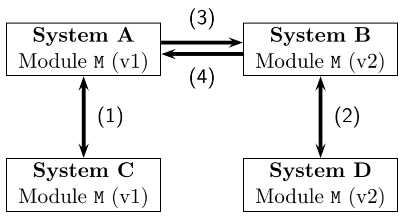
```
State ::= ENUMERATED {on, off, out-of-order, ...}
Description ::= SEQUENCE { surname    IA5String,
                           first-name IA5String,
                           age        INTEGER,
                           ... }
Dimensions ::= SET { x INTEGER,
                     y INTEGER,
                     ... }
Afters ::= CHOICE { cheese  IA5String,
                    dessert IA5String,
                    ... }
```

The `INTEGER` and `BIT STRING` types, whose syntax is very much similar to the `ENUMERATED` type's are implicitly extensible since their naming list is not restrictive: provided these types are not constrained (but subtype constraints can be extensible as well), one may use any unnamed integer (see on page 132) and any unnamed-bit position (see on page 146).

In a new version of the specification, the specifier can then include new components after the "..." marker. The types (and the module in which they are defined) keep the same name:

```
State ::= ENUMERATED {on, off, out-of-order, ...,
                      stand-by}            -- version 2
Dimensions ::= SET { x INTEGER,
                     y INTEGER,
                     ...,
                     z INTEGER }           -- version 2
Afters ::= CHOICE { cheese  IA5String,
                    dessert IA5String,
                    ...,
                    coffee  NULL }         -- version 2
Afters ::= CHOICE { cheese  IA5String,
                    dessert IA5String,
                    ...,
                    coffee  NULL,          -- version 2
                    cognac  IA5String }    -- version 3
```

In this last definition, note that once the type includes an extension marker, it is indefinitely extensible and that it is useless to insert a new "..." marker after each new extension.



	If System	**sends data to System**	**then**
(1)	**A** **C**	**C** **A**	everything goes well because both **A** and **C** have the same version (v1) of the ASN.1 specification
(2)	**B** **D**	**D** **B**	everything goes well because both **B** and **D** have the same version (v2) of the ASN.1 specification
(3)	**A**	**B**	**B** does not receive all the data expected but should be able to recover
(4)	**B**	**A**	**A** receives too much data but should be able to ignore them

Figure 12.1: Interworking using extensibility

The process of decoding is detailed in Figure 12.1 for the various cases of specification versions. When decoding an extensible type value, we should detect that:

- the expected extensions are absent or
- unexpected extensions are present[21],

using the extensions that can be new components of a `SEQUENCE` or `SET` type, new alternatives of a `CHOICE` type, new states for an `ENUMERATED` type, and even new values or new lengths in the case of extensible constraints (see Section 13.12 on page 291).

In such cases, the decoder should not stop to generate a protocol error but should switch to a dedicated action specified by the application designer: in general, this consists in ignoring the unexpected extensions or affect a default value to the expected extensions; it may also inform the application using a signal as agreed beforehand.

This particular signal for trouble report is called *exception*. It is specified in the abstract syntax by an exception marker "`!`"; it informs

[21]The updating of a type from one version to the next one is obviously carried out by inserting the extensions *at the end of the type definition* (or just before the second marker if present) and not by inserting them between the extensions of the previous version.

the application of the problems encountered and specifies the reason for them. Of course, it does not fall within the competence of ASN.1 to specify what the application should do when informed of this exception; this action, which can be of various nature (ignore the unnecessary or missing elements, re-send the message 'as is' for a relay application, display a message on the user interface, replace the missing elements by default values...), depends on the protocol strategy (which can be specified in SDL, for example, see Section 23.1 on page 476). Note that the ASN.1 specifier needs not insert an exception marker in a specification if no action is associated with it by the protocol designer.

An exception appears immediately after the extension marker "..." or in some subtype constraints (see rule ⟨3⟩ on page 293). It should be inserted in the first version of the ASN.1 specification. It is by default an integer value (and, in this case, the exception list and the meaning of the exceptions should be provided with the ASN.1 specification), otherwise its type should be declared as an open type (see on page 343), i.e. the type should be indicated before the exception value and separated with the ":" symbol:

```
Description ::= SEQUENCE {
  surname    IA5String,
  first-name IA5String,
  age        INTEGER,
  ...!extended-description }
extended-description INTEGER ::= 1

Dimensions ::= SET { x INTEGER,
                     y INTEGER,
                     ... !IA5String:"dimension error"}

Afters ::= CHOICE { cheese  IA5String,
                    dessert IA5String,
                    ...!ExtensionPb:greedy,
                    coffee  NULL,
                    cognac  IA5String }
ExtensionPb::= ENUMERATED {greedy, ...}
```

Other examples can be found in the ROSE ASN.1 specification [ISO13712-1] where all the exceptions have a self-explicit name (such as `unrecognizedOperation`, for example), which is an identifier taken in a named integer list after an `INTEGER` type.

The exception marker is particularly recommended if the abstract notation is used by a relay application in charge of handing over the information: this application should be indicated to re-emit[22] the octet stream corresponding to the extensions (connections (1) and (2) on Figure 12.2 on the following page) even if it is based on some older version of the specification (i.e. even if it cannot decode them to re-encode them afterwards).

In all the examples presented since the beginning of this section, the types were extensible at the end (just before the closing curly bracket). In late 1995, on proposal of several working groups, the ASN.1 standard was amended to allow a second extension marker "..." (the notation with only one marker remains nonetheless valid):

```
T ::= SEQUENCE { a    A,
                 b    B,
                 ...,
                 ...,
                 c    C }
```

The extension root is then decomposed into two parts (until the first extension marker, and then after the second one). Extensions are inserted, one after the other, before the second extension marker that may appear anywhere in a `SEQUENCE`, `SET` or `CHOICE` type. These types remain nevertheless extensible only in one place at a time (namely immediately before the second marker), i.e. there can be at most two extension markers in a constructed type.

Besides, when several extensions are added in a `SEQUENCE` or `SET` type and if some of them are marked `OPTIONAL` or `DEFAULT`, the decoder cannot infer the difference between an optional extension for which no value was provided and a non-optional extension that was not taken into account in the version of the sending application. To tackle the problem, we can delimit the different versions of the `SEQUENCE`, `SET` and `CHOICE` types with version double square brackets "`[[`" and "`]]`":

```
Afters ::= CHOICE {
  cheese   IA5String,
  dessert  IA5String,
  ...!ExtensionPb:greedy,
  [[coffee NULL     ]],  -- version 2
  [[cognac IA5String]] } -- version 3
```

[22]We assume the three applications use different versions of the same abstract syntax (i.e. all versions are registered with the same object identifier).

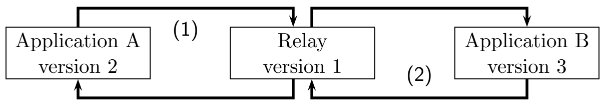


Figure 12.2: An example of relay between different extensions of the same specification

If the extension consists in adding only one new component, the version double square brackets can be omitted.

In type `T` below, if either of the mandatory components `d` or `e` appear in one of the values, the other should also appear (such a condition cannot be denoted without the version double square brackets, except when using a `WITH COMPONENTS` constraint, see Section 13.9 on page 277):

```
T ::= SEQUENCE { a    A,
                 b    B,
                 ...,
                 [[d  D,
                   e  E]],
                 ...,
                 c    C }
```

Ignored by the Basic Encoding Rules (BER), the double square brackets have the advantage of cutting down the size of the encoding when using the Packed Encoding Rules[23] (PER). The extension markers and the version brackets do not change in any way the abstract value notation of each type.

In order to ensure the tag distinctness and the decoding determinism, a few specific rules apply to the constructed types:

- if the module includes the `AUTOMATIC TAGS` clause in its header and providing no root component is already tagged by the specifier, the root components (possibly made of two parts) are tagged first before tagging the extensions in their order of appearance;

- no extension can be tagged by the specifier if no root component is tagged manually first;

[23]Some specifiers may be tempted to replace in the example `T` above the extension group (nested in version brackets) with a `SEQUENCE` type whose components would consist of these extensions. Even though this substitution has no effect on the PER encoding of the type `T` (see Section 20.6.12 on page 446), the version brackets should be preferred since they have a semantics that is independent from the encoding rules.

- the tags of the extensions in a `SET` or `CHOICE` type must be ordered according to the following *canonical order*: first, the tags of `UNIVERSAL` class, then the `APPLICATION` class tags, the `PRIVATE` class tags and finally those of context-specific class; in every class, the tag numbers should be in increasing order;

- a virtual component or alternative is added at the insertion point of the extensible `SEQUENCE`, `SET` and `CHOICE` types (at the end of the type definition if it contains a single extension marker or if the module includes the `EXTENSIBILITY IMPLIED` clause in its header, or just before the second marker otherwise) so that a BER, CER or DER decoder that would conform to a version 1 of an abstract syntax could associate the reception of an unknown component with the appropriate extension marker to trigger the dedicated exception if necessary; this virtual component's tag is different from any of those potentially defined in ASN.1 (say `[IMAGINARY 0]`); it is added once developed the `COMPONENTS OF` clauses; we may then check the condition on distinct tags according to the rules given in the previous sections of this chapter, leaving aside the extension markers and the version double square brackets. The tag distinctness is therefore not respected for a type[24] such as:

```
Person ::= SET {
  surname    [0] IA5String,
  first-name [1] IA5String,
  contact    CHOICE { phone-number   [2] NumericString,
                      e-mail-address [3] NumericString,
                      ... },
  info       CHOICE { age            [4] INTEGER,
                      ... } }
```

  since the addition of virtual components would lead to the type described on the top of the following page, which does not comply with the tag distinctness condition.

[24]This type is not defined in an automatic tagging environment (in such an environment, there is no need to test the addition of an imaginary component).

```
Person ::= SET {
  surname    [0] IA5String,
  first-name [1] IA5String,
  contact    CHOICE { phone-number   [2] NumericString,
                      e-mail-address [3] NumericString,
                      ...,
                      imaginary      [IMAGINARY 0] T },
  info       CHOICE { age            [4] INTEGER,
                      ...,
                      imaginary      [IMAGINARY 0] T} }
```

Various examples are given in [ISO8824-1, clause 47.8]. To go further, the reader may consider consider the following type, found in ITU-T recommendation H.245 about multimedia system control[25]:

```
MasterSlaveDeterminationReject ::= SEQUENCE {
  cause CHOICE { identicalNumbers NULL,
                 ...},
  ...}
```

To spare us fastidious checkings, it is recommended to include the `AUTOMATIC TAGS` clause in the module header before starting defining extensible types.

It is particularly recommended to use extension markers (with which exception markers should be associated when possible) in the specifications that are still under construction to avoid interworking problems in the future[26]. A specifier may also decide that every one of the `ENUMERATED`, `SEQUENCE`, `SET` and `CHOICE` types but those imported in the module are extensible by default. For this the clause `EXTENSIBILITY IMPLIED` should be inserted in the module header:

```
ModuleName DEFINITIONS AUTOMATIC TAGS
                       EXTENSIBILITY IMPLIED ::=
BEGIN
-- ...
END
```

Although extension markers have been very popular in specifications since 1994, we hardly ever come across the `EXTENSIBILITY IMPLIED`

[25]Indeed this type is correct in the context of the H.245 ASN.1 module, which includes the `AUTOMATIC TAGS` clause in its header.

[26]In terms of encoding the cost is very low: no extra octet when using the BER, an extra bit per extension marker for the PER.

clause. It is worthwhile mentioning its side effect for the PER encoding (see Chapter 20): an extensibility bit is systematically added before encoding all the values of `ENUMERATED`, `SEQUENCE`, `SET` and `CHOICE` types. The specifier should also make sure to insert the extension marker "..." before actually extending one of the types defined in the module that has the `EXTENSIBILITY IMPLIED` clause in its header.

When producing a new version of a module registered in the registration tree (see on page 163), its object identifier does not change if the modifications that motivated the updating were only related to the extensible types. Indeed, the concept of extension markers makes it possible to generate encoders and decoders that can manage differences between versions.

12.9.2 Reference Manual

$$\begin{aligned}
&\mathit{ExtensionAndException} \rightarrow \text{“...”}\\
&\qquad\qquad\qquad\qquad\qquad \mid\ \text{“...”}\ \mathit{ExceptionSpec}\\
&\mathit{OptionalExtensionMarker} \rightarrow \text{“,”}\ \text{“...”}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mid\ \epsilon\\
&\mathit{ExtensionEndMarker} \rightarrow \text{“,”}\ \text{“...”}
\end{aligned}$$

⟨1⟩ An extension marker "..." can appear (since ASN.1:1994) in `ENUMERATED`, `SEQUENCE`, `SET` and `CHOICE` types, in subtype constraints (see Section 13.12 on page 291), in object sets (see on page 331) or in value sets (see on page 333). It indicates that if a decoder detects extra or missing elements when comparing with those of the specification, it must not treat them as an error during the decoding process.

⟨2⟩ An `ENUMERATED`, `SEQUENCE`, `SET` or `CHOICE` type that does not include an extension marker is extensible if the module includes the `EXTENSIBILITY IMPLIED` clause in its header (see on page 114).

⟨3⟩ If the extension marker is combined with an exception marker (see production *ExceptionSpec* on page 255), this means that the decoder should execute a specific action if it receives more (or less) elements than those indicated in the type specification.

⟨4⟩ The exception marker "`!`" is not allowed in the `ENUMERATED` type at the present time (see rule ⟨6⟩ on page 139), and neither is it in the extensible information object sets or in the extensible value sets.

⟨5⟩ The *extension root* consists of the components appearing before the first extension marker "..." and of those appearing after the second one if present. The *extensions* are the components (or groups of components in version double square brackets) appearing after the extension marker when there is only one and between the two markers otherwise (see also rule ⟨6⟩ on page 294).

⟨6⟩ If there is only one extension marker, the extensions introduced for a new version of the procotol must be inserted just before the closing curly bracket.

⟨7⟩ For the `SEQUENCE`, `SET` and `CHOICE` types, if there are two extension markers, new components are inserted just before the second marker.

⟨8⟩ For the `SEQUENCE`, `SET` and `CHOICE` types, extensions may be grouped with the *version brackets* "`[[`" and "`]]`" to highlight the different versions of the specification.

⟨9⟩ A type containing an extension marker can be referenced by another type containing an extension marker, whether it is in the extension root of this type or within an extension. In such cases, the extensions are treated separately and the extensibility of a referenced type has no impact on the type that references it.

⟨10⟩ If a type defined with an extensible constraint is referenced in a subtype constraint by type inclusion (production *ContainedSubtype* on page 263), the resulting type does not inherit the extension marker. For example, if `A ::= INTEGER(0..10, ...)`, then `B ::= INTEGER(A)` is not extensible, but `C ::= INTEGER(A, ...)` is extensible.

⟨11⟩ If a type defined with an extensible constraint is constrained elsewhere by a non-extensible constraint, the resulting type is not extensible. For example, if `A ::= INTEGER(0..10, ...)`, then `B ::= A(2..5)` is not extensible, but `C ::= A` is extensible.

⟨12⟩ If a type followed by an extensible constraint is constrained elsewhere by another extensible constraint, the root of the equivalent constraint is obtained by the intersection of the two constraint roots: the equivalent constraint has the extensions of the second constraint (provided they exist) . For example, `T ::= IA5String (SIZE (1..10, ..., 15..20))(FROM ("AB", ..., "CD"))` is equivalent to `T ::= IA5String (SIZE (1..10))(FROM ("AB", ..., "CD"))`.

⟨13⟩ When an extensible structured type is subtyped by a constraint on its components introduced with the keywords `WITH COMPONENTS` (production *InnerTypeConstraints* page 277), it remains extensible.

⟨14⟩ The (explicit or implicit) presence of an extension marker in the definition of a type does not affect the value definition of this type.

⟨15⟩ The components of a `SET` or a `SEQUENCE`, or the alternatives of a `CHOICE`, which are marked `ABSENT` in an *InnerTypeConstraints* (see on page 277) cannot be present in a value even if the type is extensible.

⟨16⟩ Each time the tag distinctness is required (in extensible `SET` and `CHOICE` types, and for optional components of extensible `SEQUENCE` types), the following transformation[27] should be virtually applied before checking this condition:

– an extension marker and a virtual component (or alternative) are added at the end of the type if the module includes the `EXTENSIBILITY IMPLIED` clause in its header and if the type has no extension marker; or

– a virtual component (or alternative) is added at the end of the type if the module includes the `EXTENSIBILITY IMPLIED` clause and if the type has one extension marker; or finally

– a virtual component (or a alternative) is added just before the second extension marker.

This virtual complement is done once the `COMPONENTS OF` clauses are expanded. The virtual component (or alternative) is assumed to have a different tag, say `[IMAGINARY 0]`, from any ASN.1 type tag (not virtual), but this tag must remain the same for all the virtual components (or alternatives). If the condition on distinct tags is not respected after the addition of these virtual components (or alternatives), the type definition is not valid. An example can be found on page 251.

⟨17⟩ The previous rule needs not apply if the module includes the `AUTOMATIC TAGS` clause in its header. However using only the `EXTENSIBILITY IMPLIED` clause in the module header does exclude this previous rule (see rule ⟨10⟩ on page 114).

$$\begin{aligned} \textit{ExceptionSpec} \rightarrow\ & \text{“}\texttt{!}\text{”}\ \textit{ExceptionIdentification} \\ |\ & \epsilon \end{aligned}$$

⟨18⟩ In a complex ASN.1 specification, the decoder may have to execute a specific action if it receives more (or less) elements than those indicated in the constructed type specification or if a subtype constraint is not respected (it may happen when it is parameterized, see rule ⟨3⟩ on page 293). In such cases, the application designer should be told what

[27]This rule is meant to ensure that a BER, CER or DER decoder conforms to some earlier version of an abstract syntax, which receives an unknown component, should associate it with the appropriate extension marker and trigger the right exception. For example, this is particularly useful when extensible `CHOICE` types are components of another extensible type.

specific actions to carry out. If an *ExceptionSpec* is present, the application may undertake actions that depend on the implementation.
⟨19⟩ For the time being, the ASN.1 grammar does not systematically allow associating an exception marker to all the occurrences of the extension marker "..." (it is the case for extensible information object sets, for example).

$$\begin{array}{rcl} \textit{ExceptionIdentification} & \rightarrow & \underline{\textit{SignedNumber}} \\ & | & \underline{\textit{DefinedValue}} \\ & | & \underline{\textit{Type}} \text{ ``:'' } \underline{\textit{Value}} \end{array}$$

⟨20⟩ *DefinedValue* must reference a value of type `INTEGER`.
⟨21⟩ The third alternative denotes an exception *Value* of any *Type* (the syntax is similar to that of the production *OpenTypeFieldVal* on page 348).

Chapter 13

Subtype constraints

Contents

By the slightest constraint am I afflicted
And I solely rejoiced
In what for myself I neglected.

Tristan L'Hermite, *Les Amours*.

In the last three chapters, we have introduced ASN.1 basic and constructed types, the latter relying on the former to define more complex types. Most of the types that can be written with this material actually correspond to infinite sets of values. We now see how to restrict this set of potential values for a type.

13.1 Basics of subtyping

13.1.1 User's Guide

Many reasons may induce a specifier to use subtype constraints. First, subtyping formally refines a specification in the sense that it makes it more precise and closer to the data it models (without it, such information might have been indicated in comments but would not have been treated by ASN.1 compilers). Unfortunately, all the compilers available on the market do not generate encoders which check that the data provided by the communicating application respect the subtype constraints of the abstract syntax. As a result, every application designer has to develop and implement the adequate algorithms whereas these could be generated automatically from the ASN.1 descriptions[1].

Second, subtyping often ensures interworking[2] by improving the encoder and decoder implementation so that communicating applications (written in *C* language for the great majority) can manage their memory space more easily (when the default character string size is limited for instance).

Finally, subtyping sometimes makes it possible to generate a more compact transfer syntax. We will see in Chapter 20 that some subtype constraints are very useful for PER encoding. On the other hand, the BER encoding rules do not use them at all (see Section 18.2.17 on page 410). Note, however, that subtype constraints apply on values that comply with the abstract syntax and do not, therefore, directly influence the transfer syntax (for example, a constraint like `SIZE (4)` applied to a character string of type `UniversalString` limits the size of the encoded string to 16 octets).

[1]If the compiler generates automatically these test functions, it is recommended to remove or simply comment out these tests in the application itself to gain in processing time.

[2]Today's booming protocols such as videoconferencing (see on page 84) or the aeronautical telecommunication networks (see on page 91) in particular, rely on ASN.1 subtype constraints to permit the communicating systems to interwork more efficiently.

Generally speaking, ASN.1 subtype constraints are indicated in round brackets after the type expressions. The diversity of constraints may seem daunting at first glance; we will describe them one by one and show how to combine one with another to refine the specification as needed.

13.1.2 Reference Manual

$$\begin{array}{rcl} \textit{ConstrainedType} & \rightarrow & \underline{\textit{Type}}\ \underline{\textit{Constraint}} \\ & | & \underline{\textit{TypeWithConstraint}} \end{array}$$

⟨1⟩ All the constraints after *Type* should share at least one common value with it (see rule ⟨17⟩ on page 291).
⟨2⟩ If several *Constraint*s appear one after the other after *Type*, the possible values for this type are those of the intersection of constraints (see also rules ⟨7⟩ and ⟨8⟩ below).
⟨3⟩ In ASN.1:1990, the set of values corresponding to a constraint had to be included in the set of values of the constrained type (i.e. the type appearing before the constraint). If `A ::= INTEGER (0|1|10)`, then `B ::= A (1..10)` was not valid (see also footnote 12 on page 285).
⟨4⟩ If *Type* is (literally) a *SelectionType*, it is the `CHOICE` type that is constrained (hence the constraint is ignored as explained in rule ⟨4⟩ on page 240). If *Type* is a reference to a *SelectionType*, it is the selected type that is constrained, and not the `CHOICE` type.
⟨5⟩ If *Type* is (literally) a `SEQUENCE OF` or `SET OF` type, the *Constraint* applies on the type appearing after the keywords `SEQUENCE OF` or `SET OF` (see the differences with the production *TypeWithConstraint* on pages 232 and 234). If *Type* is a reference to a `SEQUENCE OF` or `SET OF` type, the *Constraint* applies to the `SEQUENCE OF` or `SET OF` type, and not to the type that follows these keywords.
⟨6⟩ An extensible structured type that is subtyped by a constraint on its components introduced with the keywords `WITH COMPONENTS` (production *InnerTypeConstraints* page 277) remains extensible.
⟨7⟩ If a type defined with an extensible constraint is constrained further on with a non-extensible constraint, the resulting type is not extensible, For example, `A ::= INTEGER (0..10, ...)`, then `B ::= A (2..5)` is not extensible, but `C ::= A` is.

⟨8⟩ If a type followed by an extensible constraint is constrained further on with another extensible constraint, the root of the equivalent constraint is obtained by intersection of the roots of each constraint; the equivalent constraint has the extensions of the second constraint (if present). For example, `T ::= IA5String (SIZE (1..10, ..., 15..20))(FROM ("AB", ..., "CD"))` is equivalent to `T ::= IA5String (SIZE (1..10))(FROM ("AB", ..., "CD"))`.

⟨9⟩ If a type defined with an extensible constraint is referenced in a constraint by type inclusion (production *ContainedSubtype* on page 263), the resulting type does not inherit the extension marker. For example, if `A ::= INTEGER (0..10, ...)`, then `B ::= INTEGER (A)` is not extensible, but `C ::= INTEGER (A, ...)` is.

⟨10⟩ If a type is constrained several times with extensible constraints or if a type with an extensible constraint is referenced as a *ContainedSubtype* (see on page 263) in another extensible constraint, the exception marker that must be associated with the type is the marker of the outermost constraint. The *outermost constraint* is that of the highest level type in the breakdown structure of referencing. For example, if `T ::= INTEGER(0..5, ...!5)` and `U ::= INTEGER(T|10..15, ...!15, 20..25)`, then the exception associated with `U` is `15` (and not `5`).

13.2 Single value constraint

13.2.1 User's Guide

The simplest form of subtyping consists in limiting the set of values for a type to a single value. For this, this value should be written in round brackets after the type. The value should, of course, conform to the constrained type. This subtype constraint can be applied to any ASN.1 type:

```
Two ::= INTEGER (2)
Day ::= ENUMERATED { monday(0), tuesday(1), wednesday(2),
          thursday(3), friday(4), saturday(5), sunday(6) }
Wednesday ::= Day (wednesday)
FourZ ::= IA5String ("ZZZZ")
Afters ::= CHOICE {
  cheese  IA5String,
  dessert ENUMERATED { profiterolles(1), sabayon(2),
                       fraisier(3) }}
CompulsoryAfters ::= Afters (dessert:sabayon)
```

In the examples above, `Two` and `Wednesday` are types (they begin with an upper-case letter) even though they contain only one value and `FourZ` is a character string type that contains the unique string made of exactly four capital letters "`Z`". The last type describes how to constrain the type `Afters` so that one has to have a sabayon for dessert when having afters!

This form of subtyping becomes all the more interesting when constraining a type to a limited number of values listed in round brackets separated by a vertical bar (union symbol):

```
WeekEnd ::= Day (saturday|sunday)
PushButtonDial ::= IA5String ("0"|"1"|"2"|"3"|
  "4"|"5"|"6"|"7"|"8"|"9"|"*"|"#")
```

Note that the type `PushButtonDial` contains twelve strings of a *single* character (and not strings of any number of the twelve characters listed).

13.2.2 Reference Manual

SingleValue → *Value*

⟨1⟩ *Value* must be a value of the parent type[3] or of a type that is compatible with this parent type according to the semantic model of ASN.1 (see Section 9.4 on page 121).
⟨2⟩ This subtype constraint can be applied to any type.
⟨3⟩ In general, a type is constrained by a series of values separated by a union vertical bar "`|`" presented in Section 13.11.2 on page 288.

13.3 Type inclusion constraint

13.3.1 User's Guide

To constrain a type to the same set of values as another, the corresponding type reference is given in round brackets:

```
FrenchWeekEnd ::= Day (WeekEnd)
```

which means that the type `FrenchWeekEnd` is a type `Day` that contains only the values of the type `WeekEnd` (those two types were defined in the previous section). The constrained type (`Day`) and the contained (or included) type (`WeekEnd`) should obviously derive from the same type

[3]We call *parent type*, the type that appears just before the subtype constraint (up to the bracket that opens this constraint). The parent type can also include a subtype constraint itself. Hence, a parent type is specific to a given subtype constraint.

(`ENUMERATED` here) or should be compatible according to the semantic model of ASN.1 as described in Section 9.4 on page 121.

When used alone as above, the constraint by type inclusion is of little interest. It is very often used jointly with single value constraints separated by a vertical bar:

```
LongWeekEnd ::= Day (WeekEnd|monday)
```

The capital letter of `WeekEnd` and the small letter of `monday` make the distinction between a type for the former and a value for the latter.

In the following example (where the constrained and contained types both derive from the basic type `INTEGER`), there is no need to perform an intersection because both types `T1` and `T2` have an infinite set of integers[4]:

```
T1 ::= INTEGER {trois(3), quatre(4)}
T2 ::= INTEGER {one(1), two(2), three(3), four(4)}(T1)
```

and the type `T2` is therefore equivalent to `INTEGER {one(1), two(2), three(3), four(4)}`.

As mentioned in Chapter 11 about character string types, this form of subtyping can easily define specific alphabets, derived in particular from the `UniversalString` and `BMPString` types:

```
RussianName ::= Cyrillic (Level1)
```

where `Cyrillic` and `Level1` are collections of characters derived from the `BMPString` type[5] and defined in the module `ASN1-CHARACTER-MODULE` (see rule ⟨52⟩ on page 197). In fact, `RussianName` contains all the character strings that are both in the `Cyrillic` and `Level1` sets.

Before the 1994 version of the ASN.1 standard, the keyword `INCLUDES` was necessary to introduce a constraint by type inclusion:

```
FrenchWeekEnd ::= Day (INCLUDES WeekEnd)
```

The keyword has become optional since then to make combinations of subtype constraints easier to write (see Section 13.11 on page 285).

[4]Some compilers, however, operate this intersection, wrongly considering the identifiers (or even the numbers) even though this intersection has also an infinite set integers.

[5]Besides, the type `BMPString` is formally defined in the ASN.1 standard as: `BMPString ::= [UNIVERSAL 30] UniversalString (Bmp)` where the type `Bmp` is defined in the module `ASN1-CHARACTER-MODULE`.

13.3.2 Reference Manual

ContainedSubtype → *Includes* *Type*

⟨1⟩ *Type* must be derived (by subtyping or compatibility according to the rules of ASN.1 semantic model defined in Section 9.4 on page 121) from the same primitive type as the parent type.
⟨2⟩ The permitted values are those belonging to both the parent type (see footnote 3 on page 261) and *Type* (set intersection).
⟨3⟩ If *Type* includes a tag, this must be ignored.
⟨4⟩ This subtype constraint can be applied to any type but `EMBEDDED PDV`, `EXTERNAL`, `CHARACTER STRING` and the open types (see next rule).
⟨5⟩ If the parent type is an open type (the only open type in ASN.1 is *ObjectClassFieldType* when it denotes a type field, a variable-type value field or a variable-type value set field of an information object class, see rule ⟨3⟩ on page 347), the production *TypeConstraint* on page 352 must be used.
⟨6⟩ If *Type* has an extensible constraint, the resulting type does not inherit the extension marker. For example, if `A ::= INTEGER (0..10, ...)`, then `B ::= INTEGER (A)` is not extensible but `C ::= INTEGER (A, ...)` is.

Includes → `INCLUDES`
  | ϵ

⟨7⟩ The keyword `INCLUDES` was mandatory in ASN.1:1990; its being optional since ASN.1:1994 makes writing combinations of subtype constraints easier (see production *ElementSetSpecs* on page 288). It remains mandatory in the unlikely case of a *ContainedSubtype* constraint applying to a `NULL` type!

13.4 Value range constraint

13.4.1 User's Guide

The `INTEGER` and `REAL` types, as in mathematics, can be constrained to have values on an interval. The interval boundaries are separated by the symbol "`..`" and if the sign "`<`" appears on the right or left-hand side of this symbol, the interval is open on that side (i.e. the boundary does not belong to the interval):

```
Number ::= INTEGER
From3to15 ::= Number (3..15)
From3excludedTo15excluded ::= Number (3<..<15)
```

The keywords `MIN` and `MAX` denote the maximum and minimum values (or upper and lower bound) of the parent type (the one which is before the constraint round brackets):

```
PositiveOrZeroNumber ::= Number (0..MAX)
PositiveNumber ::= Number (0<..MAX)
NegativeOrZeroNumber ::= Number (MIN..0)
NegativeNumber ::= Number (MIN..<0)
```

In the case of the `REAL` type, the effective minimal and maximal values of the parent type are (mathematically speaking) decimals that may have a great number of digits if the boundary is excluded. But this is no cause for misunderstanding[6]:

```
T ::= REAL (0..<{mantissa 5,base 10,exponent 0})
U ::= T ({mantissa 2,base 10,exponent 0}..MAX)
```

In the type `U`, `MAX` stands for the greatest decimal number strictly smaller than 5. In ASN.1, the type `U` is therefore equivalent to the type:

```
U ::= REAL ({mantissa 2,base 10,exponent 0}..
           <{mantissa 5,base 10,exponent 0})
```

As ASN.1 integers may be arbitrarily long, it is recommended to constrain the type whenever it is possible in order to facilitate interworking. The specifiers whose module are encoded in PER should keep in mind that a type judiciously subtyped like `INTEGER (123456788..123456789)` is encoded on... 1 bit! It is therefore important for them not to neglect subtype constraints.

It is sometimes quite easy to confuse the types:

```
Interval ::= INTEGER {one(1), two(2)} (one..two)
Enumeration ::= ENUMERATED {one(1), two(2)}
```

But we should remember that an `ENUMERATED` type does not model a constrained list of named integers but a list of states with which a number

[6]However, the fact that decimals could be arbitrarily long may induce interworking problems and we will see in Section 13.9 on page 277 how to constrain more finely the mantissa, the base and the exponent of a `REAL` value.

is associated for encoding purpose. Besides, since no order relation is defined on the `ENUMERATED` type, it cannot be constrained by value range.

We will see in Section 13.6 that a value range constraint can also appear after the keyword `FROM` to limit the alphabet of some character strings types.

13.4.2 Reference Manual

$$\textit{ValueRange} \rightarrow \textit{LowerEndPoint} \text{ ``..'' } \textit{UpperEndPoint}$$

⟨1⟩ This subtype constraint can only be applied to `INTEGER` and `REAL` types. It can also appear in a constraint by permitted alphabet (keyword `FROM`) applied to a character string type whose alphabet has been assigned an order relation, namely `IA5String`, `NumericString`, `PrintableString`, `VisibleString` (or `ISO646String`), `UniversalString`, `UTF8String` and `BMPString` (see rule ⟨6⟩ on page 270). The use of an interval in a constraint by permitted alphabet was not allowed in the ASN.1:1990 standard.
⟨2⟩ *LowerEndPoint* cannot be greater than *UpperEndPoint*.

$$\begin{array}{lcl} \textit{LowerEndPoint} & \rightarrow & \textit{LowerEndValue} \\ & | & \textit{LowerEndValue} \text{ ``<''} \\ \textit{UpperEndPoint} & \rightarrow & \textit{UpperEndValue} \\ & | & \text{``<'' } \textit{UpperEndValue} \end{array}$$

⟨3⟩ The sign "<" is used to open the interval (i.e. to exclude the boundary).
⟨4⟩ The interval *LowerEndPoint*..*UpperEndPoint* must contain at least one value that belongs to the parent type.

$$\begin{array}{lcl} \textit{LowerEndValue} & \rightarrow & \underline{\textit{Value}} \\ & | & \texttt{MIN} \\ \textit{UpperEndValue} & \rightarrow & \underline{\textit{Value}} \\ & | & \texttt{MAX} \end{array}$$

⟨5⟩ *Value* must be a value of the parent type or of a type that is compatible with the parent type according to the semantic model of ASN.1 (see Section 9.4 on page 121).
⟨6⟩ `MIN` and `MAX` denote the minimal and maximal values allowed by the parent type (see footnote 3 on page 261).
⟨7⟩ If the parent type is `UniversalString`, `UTF8String`, `BMPString` or `IA5String` (see rule ⟨1⟩ on this page), each interval boundary of the

permitted alphabet constraint can be either one of the character references defined in the module ASN1-CHARACTER-MODULE (see rule ⟨52⟩ on page 197), or a character string cstring (or a reference to such a string) including only one character, or a *Quadruple* (see on page 196) or a *Tuple* (see on page 197) that points to a character in the character table of the type.

⟨8⟩ If the parent type is NumericString, PrintableString, VisibleString or ISO646String (see rule ⟨1⟩ on the preceding page), each interval boundary of the permitted alphabet constraint must be a character string cstring (or a reference to such a string) including only one character.

13.5 Size constraint

13.5.1 User's Guide

Particularly useful for interworking and memory space dimensioning of applications, the size constraint can limit the length of bit, octet or character strings, but also the number of elements for SEQUENCE OF or SET OF values.

A size constraint is indicated with the keyword SIZE followed by a constraint[7] (by value range or single value in particular) that conforms to the type INTEGER (0..MAX) of natural numbers:

```
Exactly31BitsString ::=  BIT STRING (SIZE (31))
StringOf31BitsAtTheMost ::=  BIT STRING (SIZE (0..31))
EvenNumber ::= INTEGER (2|4|6|8|10)
EvenLengthString ::=
  IA5String (SIZE (INCLUDES EvenNumber))
NonEmptyString ::= OCTET STRING (SIZE (1..MAX))
```

Note that between the first two types, the difference amounts to the use of a value range constraint within the size constraint of the type StringOf31BitsAtTheMost. In the last type, the keyword MAX denotes the maximum value of the INTEGER type.

In the following example:

```
ListOfStringsOf5Characters ::=
  SEQUENCE OF PrintableString (SIZE (5))
```

[7]This very constraint clustering explains the double level of round brackets in the SIZE constraint. We shall see similar cases where subtype constraints reference one another.

the constraint applies to the type that immediately precedes it, i.e. `PrintableString`, and the type `ListOfStringsOf5Characters` models a list of any number of strings, every one of which is made of exactly five characters. But if we write:

```
ListOfStrings ::= SEQUENCE OF PrintableString
ListOf5Strings ::= ListOfStrings (SIZE (5))
```

the constraint applies to the type `ListOfStrings`, which is a reference to a `SEQUENCE OF` type. Then the type `ListOf5Strings` models lists of exactly five strings, each of which has any number of characters.

To directly subtype the `SEQUENCE OF` and `SET OF` types without using the intermediate reference of the previous example, the general rules according to which subtype constraints appear in round brackets after the type has been relaxed and a special syntax has been introduced for these two types: the constraint can be inserted between the keywords `SEQUENCE` (or `SET`) and `OF` as in:

```
ListOf5Strings ::= SEQUENCE (SIZE (5)) OF PrintableString
ListOf5StringsOf5Characters ::=
  SEQUENCE (SIZE (5)) OF PrintableString (SIZE (5))
```

Until 1994, this constraint could not include ‘outside’ round brackets and was written:

```
ListOf5StringsOf5Characters ::=
  SEQUENCE SIZE (5) OF PrintableString (SIZE (5))
```

This double level of round brackets was introduced to homogenize the syntax of the various subtype constraints since other constraints have been allowed at that place since the 1994 edition (see rules ⟨3⟩ on page 232 and ⟨3⟩ on page 234).

13.5.2 Reference Manual

SizeConstraint → `SIZE` *Constraint*

⟨1⟩ *Constraint* must be a valid subtype constraint for the parent type `INTEGER (0..MAX)`. It is denoted in round brackets.
⟨2⟩ This constraint can only apply to the types `BIT STRING`, `OCTET STRING`, `SEQUENCE OF`, `SET OF`, to one of the character string types defined in Chapter 11 (these types are listed on page 192) or to the `CHARACTER STRING` type (see rule ⟨5⟩ on page 308).
⟨3⟩ The unit of measure is the bit for the type `BIT STRING`, the octet for the type `OCTET STRING`, the element for the type `SET OF` or `SEQUENCE OF` and the character for the character string types.

⟨4⟩ When the `SIZE` constraint is extensible, a definition such as `T ::= IA5String (SIZE (1|2, ..., 3))` is equivalent to `T ::= IA5String (SIZE (1|2), ..., SIZE (1|2|3))`.

$$\begin{array}{rl} \textit{TypeWithConstraint} \rightarrow & \texttt{SEQUENCE}\ \underline{\textit{Constraint}}\ \texttt{OF}\ \underline{\textit{Type}} \\ | & \texttt{SEQUENCE}\ \textit{SizeConstraint}\ \texttt{OF}\ \underline{\textit{Type}} \\ | & \texttt{SET}\ \underline{\textit{Constraint}}\ \texttt{OF}\ \underline{\textit{Type}} \\ | & \texttt{SET}\ \textit{SizeConstraint}\ \texttt{OF}\ \underline{\textit{Type}} \end{array}$$

⟨5⟩ The production *TypeWithConstraint* is explained on pages 232 and 234.

13.6 Alphabet constraint

13.6.1 User's Guide

The ASN.1 character string types (introduced in Chapter 11) generally have too large alphabet to make a real interworking possible. It is therefore recommended and sometimes even necessary to limit their alphabet, i.e. the set of characters available for writing character strings of one of those types.

We have already mentioned that, in practice, few compilers check for the conformity of transmitted values to the subtype constraints; it proves all the more relevant for the permitted alphabet constraint. However, this subtype constraint can still be very useful, for it makes the ASN.1 specification easier to understand by forbidding characters, encoded on several bytes (but are represented by the same graphical symbol in the abstract syntax) or escape characters which change the interpretation of the following character.

In order to limit the alphabet of a character string type, we use the keyword `FROM` and separate the characters by a vertical bar:

```
Morse ::= PrintableString (FROM ("."|"-"|" "))
IDCardNumber ::=
  NumericString (FROM ("0".."9"))
PushButtonDialSequence ::=
  IA5String (FROM ("0".."9"|"*"|"#"))
```

As shown in these two examples, it is possible to insert a value range constraint in a constraint by permitted alphabet (which is the reason for the double level of round brackets) provided the character string type was attributed an order relation on its alphabet. It has been so since 1994 for the types `NumericString`, `PrintableString`, `IA5String`, `TeletexString`, `T61String`, `VisibleString`, `ISO646String`, `BMPString`, `UniversalString` and `UTF8String`.

The `Morse` type above, which includes character strings made of any number of dots, dashes and spaces should not be confused with the type:

```
MorseAlphabet ::= PrintableString ("."|"-"|" ")
```

which contains only three strings of a single character[8].

Although it is allowed by ASN.1 syntax, the following definition makes no sense:

```
WrongType ::= IA5String (FROM ("Albert".."Zoe"))
```

since the canonical order mentioned in Chapter 11 is defined only on characters but not on character strings.

Besides, since 1994, it has been allowed to write:

```
Dna ::= PrintableString (FROM ("TAGC"))
```

Although the semantic interpretation is not given by the standard, the character strings of type `Dna` include any number of characters among `T`, `A`, `G` and `C`, which means that the previous type is equivalent to:

```
Dna::= PrintableString (FROM ("T"|"A"|"G"|"C"))
```

13.6.2 Reference Manual

PermittedAlphabet → `FROM` *Constraint*

⟨1⟩ *Constraint* must make up an alphabet included in that of the primitive constrained type. The constraint must be written in round brackets.
⟨2⟩ If *Constraint* does not include one-character strings, the strings made of several characters are equivalent to a union of one-character strings (i.e. separated by "`|`"). This rule did not exist in ASN.1:1990 since strings had to conform to the parent type constrained by `SIZE (1)`.

[8]Similarly, we should not confuse the type `PushButtonDialSequence` above with the type `PushButtonDial` defined on page 261. In fact, for the following examples:

```
BasicLatin-or-Arabic ::= BMPString (BasicLatin|Arabic)
BasicLatin-and-Arabic ::= BMPString (FROM (BasicLatin|Arabic))
```

a string of type `BasicLatin-or-Arabic` is either a string of type `BasicLatin`, or a string of type `Arabic`, although a string of type `BasicLatin-and-Arabic` is made of characters picked indifferently in the alphabets of type `BasicLatin` and `Arabic`. The reader may refer to a similar example for the types `CapitalAndSmall` and `CapitalOrSmall` defined on page 287.

⟨3⟩ This subtype constraint may be applied to any character string type (production *RestrictedCharacterStringType* page 192).
⟨4⟩ *Constraint* can sometimes include a character interval (see production *ValueRange* below.
⟨5⟩ When the `FROM` constraint is extensible, a definition like `T ::= IA5String (FROM ("abc", ..., "de"))` is equivalent to `T ::= IA5String (FROM ("abc"), ..., FROM ("abcde"))`.

ValueRange → *LowerEndPoint* ".." *UpperEndPoint*

⟨6⟩ The *ValueRange* constraint can appear in a *PermittedAlphabet* constraint applied to a character string type for which an order relation has been defined on its alphabet, namely `IA5String` (see rule ⟨12⟩ on page 193), `NumericString` (see ⟨15⟩ on page 193), `PrintableString` (see ⟨17⟩ on page 194), `VisibleString` or `ISO646String` (see ⟨33⟩ on page 195), `UniversalString` (see ⟨46⟩ on page 196), `UTF8String` (see ⟨46⟩ on page 196) and `BMPString` (see ⟨47⟩ on page 196).
⟨7⟩ *LowerEndPoint* must not be greater than *UpperEndPoint* according to the order relation defined on the relevant character string type (see Table 20.3 on page 444).
⟨8⟩ The use of a value range in a permitted alphabet constraint was not allowed in the ASN.1:1990 standard.

LowerEndPoint → *LowerEndValue*
| *LowerEndValue* "<"
UpperEndPoint → *UpperEndValue*
| "<" *UpperEndValue*

⟨9⟩ The symbol "<" can be used to represent a semi-open or open interval (i.e. excluding one of the boundaries or both), even though it is useless for character string types

LowerEndValue → *Value*
| `MIN`
UpperEndValue → *Value*
| `MAX`

⟨10⟩ *Value* must be a value of the parent type or a value of a type that is compatible with this parent type according to the semantic model of ASN.1 (see Section 11.14 on page 197).
⟨11⟩ If the parent type is `UniversalString`, `UTF8String`, `BMPString` or `IA5String` (see rule ⟨1⟩ on page 265), each interval boundary of the permitted alphabet constraint can be either one of the character

references defined in the module `ASN1-CHARACTER-MODULE` (see rule ⟨52⟩ on page 197), or a character string `cstring` (or a reference to such a string) including only one character, or a *Quadruple* (see on page 196) or a *Tuple* (see on page 197) which point to a character in the character table of the type.

⟨12⟩ If the parent type is `NumericString`, `PrintableString`, `VisibleString` or `ISO646String` (see rule ⟨1⟩ on page 265), each interval boundary of the permitted alphabet constraint must be a character string `cstring` (or a reference to such a string) including only one character.

⟨13⟩ `MIN` and `MAX` denote the smallest or the greatest character of the parent type according to the canonical order defined on the character string type.

13.7 Regular expression constraint

13.7.1 User's Guide

While investigating the correspondence of types between XML Schemas [W3C00] and ASN.1 (see Section 21.5 on page 458), the ASN.1 working group have acknowledged the interest of using regular expressions to define some character string subtype constraints.

The corrigendum for the ASN.1 standard [ISO8824-1DTC4] that will introduce these regular expressions is now circulating among the national bodies of ISO and ITU-T for vote.

Subtypes using regular expressions will be introduced with the keyword `PATTERN` followed by a character string; these expressions will consist of the metacharacters given in Table 13.1 on the following page. This notation is very similar to what can be found in *Perl*, XML or *Unix* with the **grep** command; for ASN.1, specific features have been added to reference the characters of the `UniversalString` alphabet.

The subtyping constraint will be satisfied if all the characters match the pattern given by the regular expression.

For example, regular expressions would make it possible to define exactly formats for dates, prices or phone numbers as in:

```
DateAndTime ::=
  VisibleString(PATTERN "\d#2/\d#2/\d#4-\d#2:\d#2")
                -- DD/MM/YYYY-HH:MM
```

Metacharacter	**Meaning**
`[ ]`	Match any character in the set where ranges are denoted by "`-`". A caret "`^`" after the opening curly bracket complements the set next to it.
`{`*g*`,`*p*`,`*r*`,`*c*`}`	Match the `UniversalString` character according to the *Quadruple* production on page 196.
`\N{`*name*`}`	Match the named character (or any character of the named-character set) as defined in the `ASN1-CHARACTER-MODULE` (see rule ⟨52⟩ on page 197).
`.`	Match any character (except one of ASN.1 newline characters matched by "`\n`").
`\d`	Match any digit (equivalent to `"[0-9]"`).
`\w`	Match any alphanumeric character (equivalent to `"[a-zA-Z0-9]"`).
`\t`	Match the horizontal tabulation character (code 9).
`\n`	Match any of the newline characters of codes 9, 10, 13 & 32 in Table 11.2 on page 178.
`\r`	Match the carriage return character (code 13).
`\s`	Match any of ASN.1 white-space characters (space, tabulations, newlines).
`\d`	Match a (alphanumeric-) word boundary.
`\`	Quote the next metacharacter and cause it to be interpreted literally.
`\\`	Match the backslash character.
`""`	Match the double-quote character (`"`).

Figure 13.1: Metacharacters for regular expressions

Metacharacter	**Meaning**
`|`	Alternative between two expressions.
`( )`	Grouping of the enclosed expression.
`*`	Match the preceding expression for zero or more occurences.
`+`	Match the preceding expression once or several times.
`?`	Match the preceding expression if present.
`#(`*n*`)`	Match the preceding expression exactly *n* times.
`#(`*n*`,)`	Match the preceding expression at least *n* times.
`#(`*n*`,`*m*`)`	Match the preceding expression at least *n* times but no more than *m* times.
`#(,`*m*`)`	Match the preceding expression no more than *m* times (and maybe 0).

Figure 13.2: Metacharacters for regular expressions (continued)

We now give a few more examples of patterns:

- the regular expression `"[\d^.-]"` matches any single digit, caret, hyphen or period.
- The regular expression `"\w+(\s\w+)*\."` matches a sentence made of at least one (alphanumeric) word.
- The regular expression `"\N{greekCapitalLetterSigma}"` matches the GREEK CAPITAL LETTER SIGMA.
- `"[\N{BasicLatin}\N{Cyrillic}\N{BasicGreek}]+"`, or equally `"(\N{BasicLatin}|\N{Cyrillic}|\N{BasicGreek})+"`, are regular expressions that match a string made of any (non-zero) number of characters taken in the three character sets specified by names defined in [ISO10646-1].

13.7.2 Reference Manual

PatternConstraint → `PATTERN` *Value*

⟨1⟩ The corrigendum to the ASN.1 standard [ISO8824-1DTC4] that will introduce regular expressions is now circulating among the national bodies of ISO and ITU-T for vote.

⟨2⟩ This subtype constraint may be applied to any character string type (production *RestrictedCharacterStringType* page 192).
⟨3⟩ *Value* must be a character string of type `UniversalString` (or a reference to such string) which contains a regular expression. Because the sets of strings of type `UniversalString` and `UTF8String` are the same, *Value* may also be a character string of type `UTF8String`.
⟨4⟩ A regular expression is a pattern that describes a set of strings whose format conforms to this pattern. The pattern is more or less the same as an arithmetic expression in which the operators are the metacharacters defined in Table 13.1 on page 272. The smallest expressions of this pattern are placeholders that stand for a set of characters.
⟨5⟩ The regular expression *PatternConstraint* selects the values of the parent type that satisfy the entire regular expression. Unlike other regular expression notations, ASN.1 does not provide the metacharacters "`^`" and "`$`" to match the beginning and the end of a string respectively: hence, values whose leading and/or trailing characters are not matched by the regular expression are not accepted, except if the latter includes "`.*`" at its beginning, at its end or both.
⟨6⟩ Most characters (except the metacharacters of Table 13.1 on page 272) are regular expressions that match themselves.
⟨7⟩ A list of characters enclosed by "`[`" and "`]`" matches any single character in that list. If the first character of the list is the caret "`^`", then the subexpression matches any character of the parent type that is not in the list. A range of characters may be specified by giving the first and last characters (according to the order relation associated with the parent type, see ⟨6⟩ on page 270) separated by a hyphen "`-`". The metacharacters of Table 13.1 on page 272 except "`]`" and "\" loose their special meaning between square brackets. The symbol "`^`" placed anywhere except in the first position (or preceded by a backslash "\") matches a literal caret. A symbol "`-`" placed immediately after the opening square bracket or immediately before the closing square bracket (or preceded by a backslash "\") matches a literal hyphen. A symbol "`]`" that follows the opening square bracket matches a literal closing square bracket.
⟨8⟩ To avoid any ambiguity between two [ISO10646-1] characters that have the same glyph (graphical symbol), the notations "{*group*,*plane*,*row*,*cell*}" (similar to the *Quadruple* grammar production defined on page 196), "`\N{`valuereference`}`" (where valuereference is a reference to one of the characters defined in the `ASN1-CHARACTER-MODULE`, see ⟨52⟩ on page 197) and "`\N{`typereference`}`"

(where typereference is a reference to one of the character sets defined in the `ASN1-CHARACTER-MODULE`) are provided (see Table 13.1 on page 272).
⟨9⟩ Two or more subexpressions may be joined by the infix operator "`|`". The resulting regular expression matches any string matched by either subexpression.
⟨10⟩ A subexpression may be followed by one of the repetition operators defined in Table 13.2 on page 273: "`?`", "`*`", "`+`", "`#(`n`,`m`)`", "`#(`n`,)`", "`#(,`m`)`" and "`#(`n`)`".
⟨11⟩ Repetition (see previous rule) takes precedence over concatenation, which in turn takes precedence over alternation with "`|`". These precedence rules may be overriden by adding round brackets ("`(`" and "`)`") around a subexpression.
⟨12⟩ When a regular expression contains subexpressions in square brackets ("`[`" and "`]`"), each opening bracket "`(`" (not preceded by a backslash "`\`") is successively assigned a number (beginning at 1) from the left to the right of the regular expression. Each subexpression can then be referenced inside a comment with a notation like "`\1`", "`\2`"... that uses the associated integer. The ASN.1 standard provides this formal notation for referencing subexpressions in comments or in text associated to an ASN.1 specification to document it, but there is no obligation to use it:

```
DateAndTime ::= VisibleString
  (PATTERN "((\d#2)/(\d#2)/(\d#4)-(\d#2:\d#2))")
    -- \1 is a date in which \2 is the month, \3 the day,
    -- \4 the year and \5 the time (in hours and minutes)
```

13.8 Constraint on `SEQUENCE OF` or `SET OF` elements

The constraints described so far applied mainly on basic types; we now come to constraints for subtyping constructed types.

13.8.1 User's Guide

As seen in Section 13.5, a `SEQUENCE OF` or `SET OF` type can be constrained inserting the constraint between the keywords `SEQUENCE` (or `SET`) and `OF`. But if the definition of such a type is imported from another ASN.1 module, we cannot refine it by insertion of a constraint which, should we remind it, must apply to the elements type (whereas a constraint that immediately follows a reference to a `SEQUENCE OF` or `SET OF` type applies to this very constructed type). We need, therefore, a constraint that could go down in the definition of the constructed type to apply on its elements type: it is the constraint `WITH COMPONENT` (without final "`S`").

Let the type:

```
TextBlock ::= SEQUENCE OF VisibleString
```

that we use for defining an address on several lines including no more than 32 characters:

```
AddressBlock ::=
  TextBlock (WITH COMPONENT (SIZE (1..32)))
```

or for defining a block that contains only digits (and spaces):

```
DigitBlock ::=
  TextBlock (WITH COMPONENT (NumericString))
```

where the `VisibleString` type of the elements of the type `TextBlock` is constrained by `NumericString`, which is a constraint by type inclusion (the keyword `INCLUDES` is omitted here). These two types are respectively equivalent to:

```
AddressBlock ::= SEQUENCE OF VisibleString (SIZE (1..32))
DigitBlock ::= SEQUENCE OF VisibleString (NumericString)
```

The keywords `WITH COMPONENT` may, of course, be followed by any subtype constraint conforming to the type of the elements of the `SEQUENCE OF` (or `SET OF`) type.

The constraints can, for example, be 'piled up' to subtype the following type:

```
IntegerMatrix ::=
  SEQUENCE SIZE (6) OF SEQUENCE SIZE (6) OF INTEGER
```

as in:

```
CoordinateMatrix ::= IntegerMatrix (WITH COMPONENT
                         (WITH COMPONENT (-100..100)))
```

which is the type of square matrices of dimension 6 whose elements are integers between -100 and 100, that is:

```
CoordinateMatrix ::= SEQUENCE SIZE (6) OF
  SEQUENCE SIZE (6) OF INTEGER (-100..100)
```

We will see in Part III on page 391 that this subtype constraint influences in no way the BER or PER encodings of a value of type `SEQUENCE OF` or `SET OF`.

13.8.2 Reference Manual

InnerTypeConstraints →
  `WITH COMPONENT` *SingleTypeConstraint*
  | `WITH COMPONENTS` *MultipleTypeConstraints*

⟨1⟩ The clause `WITH COMPONENT` is allowed only if the parent type is `SET OF` or `SEQUENCE OF`.
⟨2⟩ The clause `WITH COMPONENTS` (which applies on types like `SEQUENCE` or `SET`) is exposed in Section 13.9.2 on page 281.

SingleTypeConstraint → *Constraint*

⟨3⟩ *Constraint* is one of the subtype constraints that can be applied to the type of the elements of the parent type, i.e. the type after the keywords `SEQUENCE OF` or `SET OF` in the parent type. It is written in round brackets.

13.9 Constraints on `SEQUENCE`, `SET` or `CHOICE` components

13.9.1 User's Guide

For the `SEQUENCE` and `SET` types that include several components of generally very different types, we need to constraint the type of each component: it is the constraint `WITH COMPONENTS` (with a final "`S`").

Let the type:

```
Quadruple ::= SEQUENCE {
  alpha  ENUMERATED {state1, state2, state3},
  beta   IA5String OPTIONAL,
  gamma  SEQUENCE OF INTEGER,
  delta  BOOLEAN DEFAULT TRUE }
```

We can derive from it the following type where the component `alpha` systematically equals `state1` and the component `gamma` always has five elements[9]:

```
Quadruple1 ::=
  Quadruple (WITH COMPONENTS { ...,
                               alpha (state1),
                               gamma (SIZE (5)) })
```

This type is strictly equivalent to:

```
Quadruple1 ::= SEQUENCE {
  alpha  ENUMERATED {state1, state2, state3} (state1),
  beta   IA5String OPTIONAL,
  gamma  SEQUENCE SIZE (5) OF INTEGER,
  delta  BOOLEAN DEFAULT TRUE }
```

The symbol "..." means that we constrain some of the components of the `SEQUENCE` (or `SET`) type and do not change those which do not explicitly appear in the constraint. The symbol "..." should not be confused with the extension marker already presented in Section 12.9 on page 244 that we will discuss on page 291 when describing extensible constraints. A constraint `WITH COMPONENTS` may apply to an extensible (or extended) type as well.

For the `SEQUENCE` type, the components should be in the same order as in the type. For the `SET` type, the order may not be respected.

The constraint `WITH COMPONENTS` can also constrain, by means of the keywords `PRESENT` and `ABSENT`, the components marked `OPTIONAL` (`DEFAULT` respectively) to be mandatorily present or absent (only present respectively) when transmitted. Combining the subtype constraints, we can refine a `SEQUENCE` (or `SET`) type imported from another module:

```
Quadruple2 ::= Quadruple (WITH COMPONENTS {
                 alpha (state1),
                 beta  (SIZE (5|12)) PRESENT,
                 gamma (SIZE (5)),
                 delta OPTIONAL })
```

We have not used the "..." symbol since all the components of the `SEQUENCE` type are specified in the `WITH COMPONENTS` constraint. In

[9] Note that contrary to the keywords `WITH COMPONENT`, followed by round brackets that delimit another constraint, the keywords `WITH COMPONENTS` are followed by curly brackets to recall the structure of the `SEQUENCE` (or `SET`) type whose components are being constrained.

the component `delta`, the `OPTIONAL` marker means that no constraint applies to it (i.e. it keeps its default value). If only the presence of some particular components should be imposed (without using any other subtype constraints), the keywords `PRESENT` and `ABSENT` can be omitted, and the identifiers to be transmitted can be listed:

```
Quadruple3 ::=
  Quadruple (WITH COMPONENTS {alpha, beta, gamma})
```

Thus, the type `Quadruple3` is equivalent to:

```
Quadruple3 ::=
  SEQUENCE { alpha  ENUMERATED {state1, state2, state3},
             beta   IA5String,
             gamma  SEQUENCE OF INTEGER }
```

This form of subtyping is particularly interesting when defining 'conformance sets' derived from a common type where some of its possibilities are limited. When testing a protocol (see Section 23.2 on page 480), we can make sure that in some specific configurations of the communicating application or systems, its behavior conforms to what is expected from the specification.

We will see in Part III on page 391 that this subtype constraint does not impact the BER or PER encoding of a `SEQUENCE` or `SET` value; it is therefore an interesting means of constraining a too general abstract syntax while preserving its encoding properties (same tags for the BER, same component presence bit-field for the PER).

The constraint `WITH COMPONENTS` may also apply to the types defined with a `SEQUENCE` type, i.e. `REAL`, `EMBEDDED PDV`, `EXTERNAL`, `CHARACTER STRING`[10] and `INSTANCE OF`.

For example, it is recommended to make system interwork more easily by applying this constraint to the `REAL` type[11]:

```
ConstrainedReal ::=
  REAL (WITH COMPONENTS { mantissa (-65535..65536),
                          base     (2),
                          exponent (-127..128) })
```

[10]The constraint `WITH COMPONENTS` is also used in the standardized formal definition of the negotiation context switching types `EMBEDDED PDV`, `EXTERNAL` and `CHARACTER STRING` as we shall see in Chapter 14.

[11]Such a subtyping for the three components of a `REAL` type has only been possible since 1994, when it was formally defined with a `SEQUENCE` type.

The following is a more complex example from the CMIP standard [ISO9596-1] where the PDU `ROIV-m-Linked-Reply` of the ROSE protocol [ISO9072-2] is constrained:

```
ROIV-m-Linked-Reply-Action ::=
  ROIV-m-Linked-Reply (WITH COMPONENTS {
    invokedID        PRESENT,
    linked-ID        PRESENT,
    operation-value  (m-Linked-Reply),
    argument         (INCLUDES LinkedReplyArgument
                       (WITH COMPONENTS {
                         getResult         ABSENT,
                         getListError      ABSENT,
                         setResult         ABSENT,
                         setListError      ABSENT,
                         actionResult      PRESENT,
                         processingFailure PRESENT,
                         deleteResult      ABSENT,
                         actionError       PRESENT,
                         deleteError       ABSENT }))})
```

The constraint `WITH COMPONENTS` may also be applied to a `CHOICE` type. In this case, it forbids the selection of some alternatives or imposes the choice of one of them (when a `WITH COMPONENTS` constraint is applied to a `CHOICE` type, it can obviously include only one `PRESENT` marker).

Let us consider the type :

```
Choice ::= CHOICE { a A,
                    b B,
                    c C,
                    d D }
```

and the subtypes:

```
ChoiceCD ::=
  Choice (WITH COMPONENTS {..., a ABSENT, b ABSENT})
ChoiceA1 ::= Choice (WITH COMPONENTS {..., a PRESENT})
ChoiceA2 ::= Choice (WITH COMPONENTS {a PRESENT})
ChoiceBCD ::= Choice (WITH COMPONENTS {a ABSENT, b, c})
```

For `ChoiceCD`, only the `c` and `d` alternatives can be selected. For `ChoiceA1` and `ChoiceA2`, alternative `a` is necessarily selected. For `ChoiceBCD`, alternative `a` cannot be selected but no constraint applies to the other alternatives (hence, alternative `d` can be selected): note that the absence of keyword `PRESENT` or `ABSENT` has no default meaning when the `WITH COMPONENTS` constraint applies to the `CHOICE` type.

As seen in the examples throughout this section, the same constraint may have several possible expressions depending on whether the presence markers are omitted.

13.9.2 Reference Manual

InnerTypeConstraints →
  `WITH COMPONENT` *SingleTypeConstraint*
  | `WITH COMPONENTS` *MultipleTypeConstraints*

⟨1⟩ The constraint `WITH COMPONENTS` is allowed only if the parent type is `SEQUENCE`, `SET`, `CHOICE`, `REAL` (see rule ⟨2⟩ on page 144), `EXTERNAL` (see Figure 14.1 on page 301), `EMBEDDED PDV` (see Figure 14.3 on page 305), `CHARACTER STRING` (see Figure 14.4 on page 307) or `INSTANCE OF` (see rule ⟨4⟩ on page 358).
⟨2⟩ An extensible type constrained by `WITH COMPONENTS` remains extensible.
⟨3⟩ The production *SingleTypeConstraint* that applies to the types `SEQUENCE OF` and `SET OF` is presented in Section 13.8.2 on page 277.

MultipleTypeConstraints → *FullSpecification*
  | *PartialSpecification*

⟨4⟩ A value of the parent type appears in the subtype only if it complies with the constraints associated with every one of the components listed in *MultipleTypeConstraints*.

FullSpecification → "{" *TypeConstraints* "}"

⟨5⟩ If *FullSpecification* applies to a `SEQUENCE` or `SET` type, all the mandatory components of the type must appear in the *FullSpecification*.
⟨6⟩ If *FullSpecification* applies to a `SEQUENCE` or `SET` type, all the components marked `OPTIONAL` in the type that are not mentioned in *FullSpecification* are considered to be marked `ABSENT` by default.

PartialSpecification → "{" "..." "," *TypeConstraints* "}"

⟨7⟩ The symbol "..." is not an extension marker here (see Section 12.9 on page 244).
⟨8⟩ If *PartialSpecification* is used, no constraint is implied on the unmentioned components.

TypeConstraints → *NamedConstraint* "," $\cdots^{+}$

⟨9⟩ There may be only one *NamedConstraint* per component of the parent type, that is to say that every component can be constrained only once inside the same *MultipleTypeConstraints*.

$$NamedConstraint \rightarrow \text{identifier}\ ComponentConstraint$$

⟨10⟩ identifier must be one of the identifiers appearing in the parent type.
⟨11⟩ If the parent type is `SEQUENCE`, the identifiers must be listed in the same order as in *SequenceType*.

$$\begin{aligned} ComponentConstraint &\rightarrow ValueConstraint\ PresenceConstraint \\ ValueConstraint &\rightarrow \underline{Constraint} \\ &\mid\ \epsilon \end{aligned}$$

⟨12⟩ *Constraint* must be a subtype constraint valid for the type of the constrained component.
⟨13⟩ In the definition of a value of the constrained structured type, the component constrained by *ValueConstraint* must have a value that complies with this constraint.

$$\begin{aligned} PresenceConstraint &\rightarrow \texttt{PRESENT} \\ &\mid\ \texttt{ABSENT} \\ &\mid\ \texttt{OPTIONAL} \\ &\mid\ \epsilon \end{aligned}$$

⟨14⟩ When the parent type is `SET` or `SEQUENCE`, a component marked `OPTIONAL` in the parent type can be constrained `PRESENT` (in this case, the constraint is satisfied if the corresponding element appears in the value), `ABSENT` (in this case, the constraint is satisfied if the corresponding element is not present in the value) or `OPTIONAL`.
⟨15⟩ When the parent type is `SET` or `SEQUENCE`, a component marked `DEFAULT` in the parent type can be constrained `PRESENT` or `OPTIONAL`.
⟨16⟩ A component marked `DEFAULT` cannot be constrained `ABSENT` because the application will necessarily receive a value from the decoder, even if no value is actually transmitted.
⟨17⟩ When the parent type is `CHOICE`, a component can be constrained `ABSENT` (in this case, the constraint is satisfied if this alternative is not chosen) or `PRESENT`. There should be at most one `PRESENT` keyword in *MultipleTypeConstraints*.
⟨18⟩ In a *FullSpecification* (i.e. without "..."), an empty *PresenceConstraint* is equivalent to a `PRESENT` constraint if the parent type is `SET` or `SEQUENCE` and if the component is marked `OPTIONAL` in this parent type, but has no default semantics in other cases.

⟨19⟩ In a *PartialSpecification* (i.e. with "...”), an empty *PresenceConstraint* has no default semantics.
⟨20⟩ For the `SET`, `SEQUENCE` or `CHOICE` types, the components constrained `ABSENT` cannot be present in a value definition, even if the type is extensible.

13.10 Subtyping the content of an octet string

13.10.1 User's Guide

Many protocol developers would like to have a simple means of embedding data in the ASN.1 specification that could be encoded by rules that would differ from those used for the global specification. Of course, ASN.1 provides presentation context negociation types (introduced in Chapter 14), but this may seem heavy-handed for what they intend to do.

In the past, many specifiers have therefore turned to the type `OCTET STRING`, associating free constraints by the keywords `CONSTRAINED BY`, as in the CAMEL 03.78 protocol for GSM mobile phones, for instance) in addition to comments where those very encoding rules were described. Unfortunately, since these (informal) comments are not taken into account by compilers, they cannot generate procedure that would decode embedded values "on the fly".

The ASN.1 working group have therefore agreed to introduce a specific subtyping constraint to formalize this functionality. This constraint, introduced by the keywords `CONTAINING` and `ENCODED BY`, is the subject of the technical corrigendum [ISO8824-3DTC2], which is now circulating among the national bodies of ISO and ITU-T for vote.

For example, this new constraint would make it possible to define a value of type, say `MyType` encoded in PER, in a BER stream:

```
MoreCompact ::= OCTET STRING (CONTAINING MyType ENCODED BY
                  {joint-iso-itu-t asn1 packed-encoding(3)
                   basic(0) unaligned(1)})
MyType ::= SEQUENCE { -- ....... -- }
```

An ASN.1 compiler would then generate a BER encoder that automatically calls the PER encoder for the value of type `MoreCompact` (and similarly for the decoders).

The object identifier that follow the keywords `ENCODED BY` may also reference an ad-hoc encoding described with the new ECN encoding notation (see Section 21.6 on page 459). If the clause `ENCODED BY` is not used, the encoding rules are the same as the current module's (the `CONTAINING` may then embed in the current specification a PDU defined in another ASN.1 specification).

For example, to indicate in the ASN.1 specification that values of type `Document` must be encoded according to Acrobat PDF format, we can write:

```
Document ::= OCTET STRING (ENCODED BY pdf)
pdf OBJECT IDENTIFIER ::= { -- OID pour le codage PDF -- }
```

The ASN.1 compiler may then generate a decoder that would automatically launch the appropriate viewer (e.g. *Acrobat Reader*). Note that, in this case, the keyword `CONTAINING` is not used since there exists no ASN.1 type for describing the PDF format.

13.10.2 Reference Manual

ContentsConstraint → `CONTAINING` *Type*
 | `ENCODED BY` *Value*
 | `CONTAINING` *Type* `ENCODED BY` *Value*

⟨1⟩ The corrigendum to the ASN.1 standard [ISO8824-3DTC2] that will introduce this new subtype constraint is now circulating among the national bodies of ISO and ITU-T for vote.
⟨2⟩ This constraint can only apply to the types `BIT STRING` and `OCTET STRING`.
⟨3⟩ When a type is constrained by *ContentsConstraint*, it cannot be constrained (directly or indirectly) by any other subtype constraint.
⟨4⟩ *Value* must be a value of type `OBJECT IDENTIFIER`.
⟨5⟩ The type that is referenced after the keyword `CONTAINING` specifies that the abstract value (a bit string or an octet string) is an encoding of a value of that type.
⟨6⟩ The object identifier that is referenced after the keyword `ENCODED BY` specifies that the abstract value (a bit or octet string) is the encoding produced by the encoding rules identified by this object identifier. When this object identifier is not provided (first alternative of the *ContentsConstraint* production), the encoding rules applied to *Type* must be the same as those applied to the current ASN.1 module.

13.11 Constraint combinations

13.11.1 User's Guide

In all the examples given since the beginning of this chapter, every parent type was followed by a single subtype constraint. We now describe how we can build up intersections, unions and exclusions of constraints.

The simplest way of combining constraints is to specify them one after the other after the parent type. In this case, the resulting type has its values in the intersection[12] of the set of values corresponding to each constraint. Then, it is possible to write:

```
PhoneNumber ::=
  NumericString (FROM ("0".."9"))(SIZE (10))
```

for the type of French telephone numbers or

```
Row ::= SEQUENCE OF INTEGER
CoordinateMatrix ::= SEQUENCE SIZE (6) OF
    Row (SIZE (6))(WITH COMPONENT (-100..100))
```

for the type of square matrices of dimension 6 whose elements are integers between -100 and 100 (we may compare this type to its equivalent defined on page 276), or else

```
TextBlock ::= SEQUENCE OF VisibleString
Address ::=
  TextBlock (SIZE (3..6))(WITH COMPONENT (SIZE (1..32)))
```

for the addresses of 3 to 6 lines of 1 to 32 characters each (the first `SIZE` constraint applies to the `SEQUENCE OF` type and the second to the `VisibleString` type of each element).

We have already used the vertical bar "`|`" to produce enumerations of simple constraints (single value or value range) as in:

```
PushButtonDialSequence ::=
  IA5String (FROM ("0".."9"|"*"|"#"))
```

[12] ASN.1:1990 imposed the set of values of a constraint to be a subset of the set of values of the parent type, which involves that the definitions:

```
T1 ::= INTEGER (1..10)     or     T2 ::= INTEGER (1..10)(5..15)
U ::= T1(5..15)
```

were not semantically correct since the interval [5;15] is not included in the interval [1;10] (see also rule ⟨3⟩ on page 259). Since 1994, the types `U` and `T2` have been semantically equivalent to:

```
T ::= INTEGER (5..10)
```

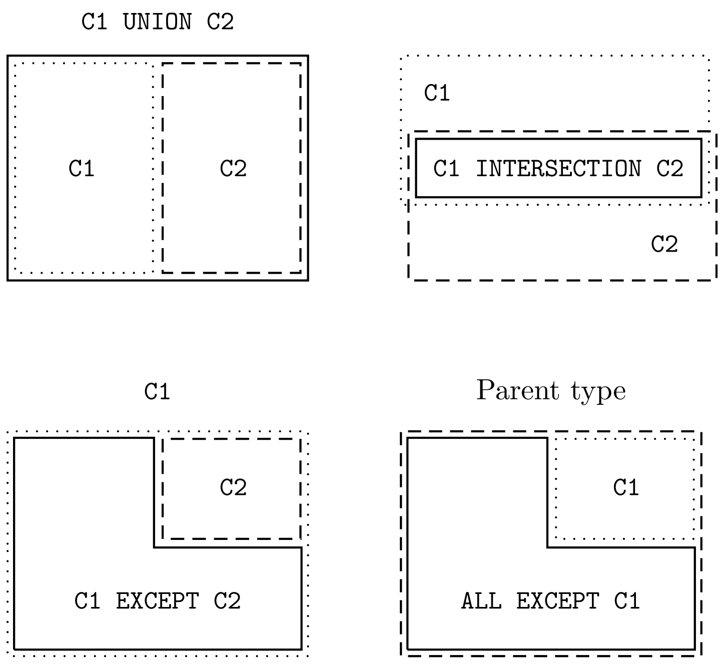


Figure 13.3: Set operators (on each diagram, the resulting set is in full line)

Since 1994, set operators have been available for combining subtype constraints. These operators, whose principle is exposed by Venn's diagrams on Figure 13.3, are:

- "`C1 UNION C2`" or "`C1 | C2`" gives the union of the set of values of constraint `C1` and that of constraint `C2` (i.e. the set of the values that belong to one of the two sets at least);
- "`C1 INTERSECTION C2`" or "`C1 ^ C2`" whose result is the intersection of the two sets of values (i.e. values that belong to both sets);
- "`C1 EXCEPT C2`" gives the complementary set of `C2` in `C1` (i.e. the values of `C1` that do not belong to `C2`) ;
- "`ALL EXCEPT C1`" gives the complementary set of `C1` in the set of values of the parent type (i.e. the values of the parent type that do not belong to `C1`).

In order to transmit Georges Perec's *Void*[13] quoted on page ii at the beginning of this book, one may use the type[14]:

```
Lipogramme ::=
  IA5String (FROM (ALL EXCEPT ("e"|"E")))
```

The set operators can also be easily used to describe an alphabet as a `UniversalString` subset:

```
SaudiName ::= BasicArabic (SIZE (1..100) ^ Level2)
ISO-10646-String ::= BMPString (FROM
    (Level2 ^ (BasicLatin | HebrewExtended | Hiragana)))
KatakanaAndBasicLatin ::=
  UniversalString (FROM (Katakana | BasicLatin))
```

As clearly shown in the previous example, a precedence is defined on the operators (`EXCEPT` priority is higher than `INTERSECTION`, which itself takes over `UNION`) and round brackets enables the user to change this priority (and it is actually the only case where the ASN.1 grammar allows to change the default priority using round brackets).

Concerning the combination of permitted alphabet constraints, one should keep in mind that a type made indifferently of capital and small letters:

```
CapitalAndSmall ::= IA5String (FROM ("A".."Z"|"a".."z"))
```

which contains a string like `"Example"`, is not equivalent to a type made exclusively of capital letters or exclusively of small letters:

```
CapitalOrSmall ::=
  IA5String (FROM ("A".."Z")|FROM ("a".."z"))
```

which contains strings like `"example"` or `"EXAMPLE"`. A quite similar example can be found in the footnote 8 on page 269.

It is also possible (and certainly quite exotic indeed!) to write the type of all strings made of one to four characters or of any number of one of the three small letters "`a`", "`b`" or "`c`":

```
ExoticString ::= IA5String (SIZE (1..4)|FROM ("abc"))
```

that is to say, this type includes strings like `"#"`, `"jug"`, `""` (the empty string), `"a"`, `"aabbcc"`... but not `"jujube"` or `"cabal"`. Unlikely though it might be, this example clearly shows that set operators potentially

[13] http://www.anatomy.usyd.edu.au/danny/book-reviews/h/A_Void.html

[14] We suggest the reader try and practice constraint combinations designing an ASN.1 type to transmit the textbook *The Exeter Text* by the same Georges Perec (translated by Ian Monk), in which the only vowel used is 'E'!

apply to any subtype constraints; the problem lies in the interpretation of the constraints in terms of sets of values to actually determine the possible values of the resulting type. The designer should be aware, however, that ASN.1 compilers are not likely to check for such excentric constraint combinations. It is therefore recommended to combine these as simply and straightforwardly as possible.

The Remote Operation Service Element standard ROSE [ISO13712-1] defines the type:

```
InvokeId ::= CHOICE { present INTEGER,
                      absent  NULL }
```

which is re-used and constrained in the DAP protocol of the directory [X.519]:

```
DAP-InvokeIdSet ::= InvokeId (ALL EXCEPT absent:NULL)
```

where the `absent` alternative is excluded[15].

Let us give a last example to convince the reader, if need be, of the power and flexibility of set operators. Let the type:

```
Identifications ::= SEQUENCE {
  idNumber  NumericString (FROM (ALL EXCEPT " "))
                          (SIZE (6)) OPTIONAL,
  telephone NumericString (FROM (ALL EXCEPT " "))
                          (SIZE (13)) OPTIONAL }
```

from which we can derive the type of a person, described by its name and either its ID number or its telephone number (we cannot use both identifications for the same person, but one of them should be present):

```
Person ::= SEQUENCE {
  name  PrintableString (SIZE (1..20)),
  ident Identifications (WITH COMPONENTS {idNumber}
                        |WITH COMPONENTS {telephone}) }
```

13.11.2 Reference Manual

ElementSetSpecs → *RootElementSetSpec*
| *RootElementSetSpec* “,” “...”
| *RootElementSetSpec* “,” “...” “,” *AdditionalElementSetSpec*

[15]This type is semantically equivalent to:

```
DAP-InvokeIdSet ::= InvokeId (WITH COMPONENTS {present PRESENT})
```

⟨1⟩ The extension marker "`...`" is presented in Section 13.12.2 on page 293.

RootElementSetSpec → *ElementSetSpec*
AdditionalElementSetSpec → *ElementSetSpec*

⟨2⟩ The set *ElementSetSpecs* consists of the union of the set of values corresponding to *RootElementSetSpec* and the set corresponding to *AdditionalElementSetSpec*.
⟨3⟩ For *RootElementSetSpec*, and possibly for *AdditionalElementSetSpec*, the set operators (`UNION`, `INTERSECTION`, `ALL` and `EXCEPT`) are applied on the extension root of each constraint ignoring the extension marker "`...`" and the extensions.
⟨4⟩ The set *ElementSetSpecs* must contain at least one value. The result must have at least one common value with the parent type, or with the parent type's root if it includes an extensible subtype constraint (see also rule ⟨17⟩ on page 291). Nevertheless, intermediate results of the set combination may be empty.
⟨5⟩ It is recommended not to write too complicated constraint combinations since compilers may not check whether these are valid.
⟨6⟩ When a constraint combination using set operators has to be extensible, the extension marker "`...`" must be placed in the global constraint at the outermost level (see rules ⟨4⟩ on page 268 and ⟨5⟩ on page 270).
⟨7⟩ If an exception marker "`!`" is necessary (see rule ⟨3⟩ on page 293), it is introduced with the production *Constraint* (see on page 293).
⟨8⟩ When an exception marker "`!`" (see production *ExceptionSpec* page 255) is present in a constraint by type inclusion (production *ContainedSubtype* on page 263) used in a set combination, this is ignored and is not inherited.

ElementSetSpec → *Unions*
| `ALL` *Exclusions*

⟨9⟩ The resulting set is made of all the values specified in *Unions* (1st alternative) or of all the values of the parent type except those specified in *Exclusions* (2nd alternative) (see Figure 13.3 on page 286).

Unions → *Intersections*
| *UElems UnionMark Intersections*

⟨10⟩ The resulting set is made of all the values specified in *Intersections* (1st alternative), or of all those appearing at least once in *UElems* or in *Intersections* (2nd alternative) (see Figure 13.3 on page 286).

UElems → *Unions*

UnionMark → "`|`"
  | `UNION`

⟨11⟩ It is recommended to use, throughout a specification, either the keywords `UNION` and `INTERSECTION`, nor the symbols "`|`" and "`^`".

Intersections → *IntersectionElements*
  | *IElems IntersectionMark IntersectionElements*

⟨12⟩ The resulting set is made of all the values specified in *IntersectionElements* (1st alternative), or of all those appearing at least once in *IElems* and in *IntersectionElements* (2nd alternative) (see Figure 13.3 on page 286).

IElems → *Intersections*

IntersectionMark → "`^`"
  | `INTERSECTION`

IntersectionElements → *Elements*
  | *Elems Exclusions*

⟨13⟩ The resulting set is made of all the values specified in *Elements* (1st alternative), or of all those appearing in *Elems* but not in *Exclusions* (2nd alternative) (see Figure 13.3 on page 286).

Elems → *Elements*

Exclusions → `EXCEPT` *Elements*

⟨14⟩ `EXCEPT` precedes over `INTERSECTION` and "`^`" which, themselves, take precedence over `UNION` and "`|`". Round brackets can be used to modify the priority.
⟨15⟩ It must be ensured that the 'negative operator' `EXCEPT`, used jointly with a recursive type, does not give an empty set of values (see rule ⟨31⟩ on page 335).

Elements → *SubtypeElements*
  | *ObjectSetElements*
  | "`(`" *ElementSetSpec* "`)`"

⟨16⟩ The production *ObjectSetElements* cannot be used when defining subtype constraints; it is introduced in Section 15.5.2 on page 331.

SubtypeElements → *SingleValue* | *ContainedSubtype*
| *ValueRange* | *PermittedAlphabet*
| *SizeConstraint* | *TypeConstraint*
| *InnerTypeConstraints*
| *PatternConstraint*

⟨17⟩ Before applying set operators (according to rule ⟨3⟩ on page 289), it must be ensured that every constraint (*SubtypeElements*) appearing in the combination *ElementSetSpecs* has at least one common value with the parent type.

13.12 Constraint extensibility

13.12.1 User's Guide

The extensibility question has already been exposed in Section 10.4 on page 135 for the `ENUMERATED` type and in Section 12.9 on page 244 for the `SEQUENCE`, `SET` and `CHOICE` types, so we will not discuss it here any further since it applies likewise[16] on extensible subtype constraints.

A constraint is extensible if it contains an extension marker "..." as in:

```
A ::= INTEGER (0..10, ...)
```

In a new version of our specification, we may write:

```
A ::= INTEGER (0..10, ..., 12)
```

Note that contrary to the constructed types `SEQUENCE`, `SET` and `CHOICE`, there is neither a second extension marker, nor version double square brackets "`[[`" and "`]]`" to indicate the different possible extensions. Moreover, if the `EXTENSIBILITY IMPLIED` clause is present in the module header, it does not alter the subtype constraints.

The set of (abstract) values of a subtype defined with an extensible constraint is simply given by the union of the set of values of the extension root and the set of values of the various extension additions. The extended `A` type above contains the integers from 0 to 10 and the integer 12 as the notation clearly indicates. Even when the subtype constraints

[16] However, if a type includes an extensible subtype constraint, it is its set of values that is extensible but not the structure of its values whereas if a constructed type (`SEQUENCE`, for example) is extensible, its own structure may be modified by adding new components.

are complex, they can be interpreted using the associated set of values. But we will see in Part III that a BER or PER encoding procedure for the type `INTEGER (0..10, ...)` knows how to encode the 'unknown' extension `12`.

If the extension marker appears in a size constraint as in:

```
S ::= IA5String (SIZE (1..10, ...))
```

it means the decoder should expect string that may have more than 10 characters.

Generally speaking, it is recommended for that matter to insert an exception marker "`!`" (presented on page 247) in all the extensible constraints to inform clearly the receiving application about values that would not conform to the extension root of the constraint:

```
E ::= INTEGER (1..10, ...!Exception:too-large-integer)
Exception ::= ENUMERATED {too-large-integer, ...}
```

This exception marker proves particularly useful when it applies to the size constraint, as for the previously defined type `S` for which the receiving application may not have dimensioned the memory space properly for receiving strings longer than 10 characters.

As discussed more thoroughly in Section 17.3 on page 389 when we come to parameterization, if a subtype constraint includes a parameter of the abstract syntax, then this is implicitly extensible since every instance of the parameter (i.e. the value to be affected to it) should modify the set of values of the constraint. In this case, it is recommended to use an exception marker:

```
ImplementedUnivStr{UniversalString:Level} ::=
  UniversalString (FROM ((Level UNION BasicLatin))
                       !characterSet-problem)
characterSet-problem INTEGER ::= 4
```

Finally, if a type piles up subtype constraints, several of which have an exception marker, it is the outermost exception that will be triggered:

```
T ::= INTEGER (0..10, ...!10)
U ::= T (2..6, ...!6)
```

If a value of type `U` is received outside the interval `2..6`, the exception `6` is triggered.

Likewise, for the type :

```
ImplementedUnivStgLevel1{UniversalString:ImplementedSubset}
  ::= UniversalString (ImplementedUnivStr{{Level1}}
        INTERSECTION ImplementedSubset, ...!level1-problem)
level1-problem INTEGER ::= 5
```

it is the outermost exception `level1-problem` (and not `characterSet-problem`) that is triggered.

We will also see in Section 13.13 that it is recommended to add an exception marker in case a user-defined constraint (introduced by `CONSTRAINED BY`) is used.

13.12.2 Reference Manual

Constraint → "`(`" *ConstraintSpec* *ExceptionSpec* "`)`"

⟨1⟩ The production *ExceptionSpec* is presented on page 255.
⟨2⟩ When a set combination of constraints should be extensible, the extension marker "`...`" must be placed in the outermost constraint (see the rules ⟨4⟩ on page 268, ⟨5⟩ on page 270 and ⟨20⟩ on page 334).
⟨3⟩ Unless it is used jointly with an extension marker "`...`", the exception marker *ExceptionSpec* can appear in a subtype constraint only if *ConstraintSpec* includes a formal parameter *DummyReference* (see on page 386) or if *ConstraintSpec* is an *UserDefinedConstraint* introduced by `CONSTRAINED BY` (see on page 296).
⟨4⟩ If a type is constrained several times with extensible constraints or if a type with an extensible constraint is referenced as a *ContainedSubtype* (see on page 263) in another extensible constraint, the exception marker that should be associated with the type is the marker of the outermost constraint. For example, if `T ::= INTEGER(0..5, ...!5)` and `U ::= INTEGER(T|10..15, ...!15, 20..25)` then the exception associated with `U` should be `15` (and not `5`).

ConstraintSpec → *ElementSetSpecs*
| *GeneralConstraint*

⟨5⟩ When an exception marker "`!`" (see production *ExceptionSpec* on page 255) appears in a constraint by type inclusion (see production *ContainedSubtype* on page 263) used in a set combination (i.e. using the `UNION`, `INTERSECTION`, `ALL` and `EXCEPT` operators), this is ignored and is not inherited.

ElementSetSpecs → *RootElementSetSpec*
| *RootElementSetSpec* "," "..."
| *RootElementSetSpec* "," "..." ","
AdditionalElementSetSpec

⟨6⟩ The extension marker "..." indicates that the reception of additional elements or their absence when comparing with those defined in the specification must not be treated as an error when decoding. The *extension root* appears before the extension marker "...". The *extensions* appear after this marker.
⟨7⟩ When producing a new version of the module, the extension insertion point is after the extension marker "...", at the end of *ElementSetSpecs*.
⟨8⟩ Contrary to extensions in `CHOICE`, `SEQUENCE` and `SET` types, the version double square brackets "`[[`" and "`]]`" together with the second extension marker "..." are not allowed in *ElementSetSpecs*, nor is *ElementSetSpecs* extensible by default if the module contains the clause `EXTENSIBILITY IMPLIED` in its header.
⟨9⟩ The rules ⟨7⟩, ⟨8⟩ and ⟨9⟩ on page 259 apply to extension markers in subtype constraints.

13.13 User-defined constraint

13.13.1 User's Guide

Though very powerful, the ASN.1 subtyping mechanism cannot always formally represent all the constraints that the specifier may want to indicate in the module. Since 1994, Part 3 of the ASN.1 standard [ISO8824-3] has defined the notion of *user-defined constraint*.

Introduced by the keywords `CONSTRAINED BY`, it can be seen as a special kind of comments even though it may induce particular treatments for encoding or decoding. In the ASN.1/*C*++ interface [TMF96] for example, a function designed by the programmer can be invoked just before encoding a value of a type that has a user-defined constraint and just after decoding it. This function can check that the informal constraints are respected but operations on the data may also be carried out.

This form of constraint can sometimes stand for some semantic links that were formerly (not *formally*) defined with macros (presented in Chapter 16). It can also indicate some complex constraint between components of a `SEQUENCE` or `SET` type, for example, whereas these used to be defined in comments. There certainly are still comments[17] after the CONSTRAINED BY clause, but the latter can require the compiler to insert a call to a user-defined function in the encoder or decoder body together with the sketch of this function in the generated file.

The following type comes from the Presentation layer protocol [ISO8823-1]:

```
PDV-List ::= SEQUENCE {
  transfer-syntax-name Transfer-syntax-name  OPTIONAL,
  presentation-context-identifier
    Presentation-context-identifier,
  presentation-data-values CHOICE {
    single-ASN1-type [0] ABSTRACT-SYNTAX.&Type
      (CONSTRAINED BY {-- Type corresponding --
       -- to presentation-context-identifier --}),
    octet-aligned    [1] IMPLICIT OCTET STRING,
    arbitrary        [2] IMPLICIT BIT STRING } }
```

and another from the remote operation service element ROSE [ISO13712-1] (in which the exception marker should be noted):

```
Reject ::= SEQUENCE {
  invokeId  InvokeId,
  problem   CHOICE {
    general      [0] GeneralProblem,
    invoke       [1] InvokeProblem,
    returnResult [2] ReturnResultProblem,
    returnError  [3] ReturnErrorProblem }}
(CONSTRAINED BY  {-- must conform to the above --
   -- definition --} ! RejectProblem:general-mistypedPDU)
```

The user-defined constraint very often depends on parameters, in which case these are indicated between the curly brackets of the CONSTRAINED BY clause. Any category of (a possibly governed) parameter is allowed; we will come back to it in Chapter 17. Nonetheless, here is an example extracted from the ASN.1 standard where the user-defined constraint is the result of the encryption of a value whose type cannot be known beforehand, and which, therefore, should be parameterized:

```
Encrypted{TypeToBeEnciphered} ::= BIT STRING (CONSTRAINED
  BY {-- must be the result of the encipherment --
      -- of some BER-encoded value of --
      TypeToBeEnciphered} !Error:securityViolation)
Error ::= ENUMERATED {securityViolation}
```

[17]Generally, compilers ignore them right from the scanning stage [ASU86].

This example also shows that a user-defined constraint generally goes along with an extension marker "`!`" to lift the errors up to the application.

13.13.2 Reference Manual

UserDefinedConstraint → `CONSTRAINED BY`
"{" *UserDefinedConstraintParameter* "," ···* "}"

⟨1⟩ Comments (nested in double dashes "`--`") can be inserted in the *UserDefinedConstraintParameter* list to precise their meaning. These can obviously not be exploited by tools.
⟨2⟩ A user-defined constraint can be seen as a special kind of comments that cannot be treated automatically. It can be used, however, by an ASN.1 tool (a compiler, for example) to ask the user to check the constraint manually or force the encoder or decoder to call a test function provided by the programmer.

UserDefinedConstraintParameter[18] →
Governor ":" *Value*
| *Governor* ":" *ValueSet*
| *Governor* ":" *Object*
| *Governor* ":" *ObjectSet*
| *Type*
| *DefinedObjectClass*

⟨3⟩ The governor and the governed parameter must conform to the compatibility rules of ASN.1 semantic model defined in Section 9.4 on page 121.
⟨4⟩ *Value*, *ValueSet*, *Object*, *ObjectSet*, *Type* and *DefinedObjectClass* may (but need not) be *ActualParameter*s in the *ParameterList* (see on page 385) of the *ParameterizedAssignment* that is being defined. In this case, the governor, when present, must appear both in the *ParameterList* of the current type assignment and in the *UserDefinedConstraintParameter* list of the user-defined constraint.
⟨5⟩ The *Governor* of a *Value* or *ValueSet* must be a type.
⟨6⟩ The *Governor* of an information object or an information object set must be a reference to an information object class.

Governor → *Type*
| *DefinedObjectClass*

[18]This grammar production has been slightly changed by [ISO8824-3DTC1].

Chapter 14

Presentation context switching types

Contents

Ambiguity, however, is often related to a lack of linguistic or 'situational' context [...]. But in order to build up its system upon a priori definitions and combination rules, the object which is dealt with should be like that of the mathematician: self-referencial.

Claude Hagège, *Generative grammar.*

We now introduce three types very different from those we have dealt with so far. By changing the presentation context[1] (6th layer of the OSI model), they enable the user to embed in a data flow values that may not necessarily conform to an abstract syntax specified in ASN.1 or be encoded according to the same transfer syntax as the one used for transmitting the PDV (the highest-level value) of our specification. This

[1]A presentation context is the association of an abstract syntax with a transfer syntax.

new abstract and/or transfer syntax(es) can be dynamically negotiated between two communicating applications.

These types are employed for specifying relay systems such as an e-mail application that would relay an message with a presentation context identical to the one used by the sender, or a file system (equivalent to the Unix file transfer protocol `ftp`) that would provide a file with the same presentation context as used when downloading it from the database. We shall see that one of them makes it possible to embed character strings that do not conform to any of the numerous ASN.1 character strings.

In order to start this chapter on fairly firm ground, we suggest the reader to go back over Section 3.2 on page 20 (and particularly on Figure 3.2 on page 22) where we presented the concept of negotiation on the Presentation layer of the OSI model. In footnote 12 on page 15, we have mentioned the frequent confusion between transfer syntax and encoding rules: this confusion will be pursued throughout this chapter for the sake of the discourse.

14.1 The `EXTERNAL` type

14.1.1 User's Guide

`EXTERNAL` is historically the first type that enabled the user to change the presentation context. As it self-explicitly indicates, it models values that are external to the current specification in the sense that they are defined with another abstract syntax (specified in ASN.1 or any other abstract notation) or encoded with a transfer syntax different from that of the active presentation context.

In order to encode these embedded values properly, we must have a means of denoting their own abstract and/or transfer syntax(es). It is down to the encoder and the decoder to take into account the switch of active context. Indeed, the Presentation layer protocol may be able to negotiate a whole set of defined contexts, but the actual selection of the active context is operated locally, when emitting for the encoder and receiving for the decoder.

Until 1994, a value of `EXTERNAL` type conformed to the following structured type:

```
EXTERNAL ::= [UNIVERSAL 8] IMPLICIT SEQUENCE {
  direct-reference      OBJECT IDENTIFIER OPTIONAL,
  indirect-reference    INTEGER OPTIONAL,
  data-value-descriptor ObjectDescriptor OPTIONAL,
  encoding              CHOICE {
    single-ASN1-type      [0] ANY,
    octet-aligned         [1] IMPLICIT OCTET STRING,
    arbitrary             [2] IMPLICIT BIT STRING }}
```

The component `direct-reference` identifies the data type syntax (it is not necessarily an abstract syntax described in ASN.1); in this case, we assume the transfer yntax has been agreed on between the two applications and is therefore not negotiated on the Presentation layer.

`indirect-reference` is an integer that references one of the presentation contexts (the association of an abstract syntax with a transfer syntax) that were negotiated. This indirect reference does not guarantee that the presentation context is preserved when going through a relay application because the integer that identifies the presentation context can change between connections.

At least one of the two previous identifiers is compulsory.

The component `data-value-descriptor` is a string that describes the abstract syntax of the data, but is hardly ever used in practice.

For embedding the value in the `encoding` component, we choose the alternative:

- `single-ASN1-type` if the abstract syntax is an ASN.1 type and if the data is encoded with the same transfer syntax as the active presentation context;

- `octet-aligned` for a value that encodes in a whole number of octets, if the abstract syntax is not described in ASN.1 or if the actual transfer syntax differs from the specification's transfer syntax;

- `arbitrary` otherwise.

The type `EXTERNAL` is used, for example, in the PDUs of the Association Control Service Element (ACSE, [ISO8650-1]) invoked by all the applications that use the OSI stack:

```
Association-information ::= SEQUENCE OF EXTERNAL
```

This information necessarily belongs to another abstract syntax: the very one for which the association was required. And if the data type has been defined outside the ACSE standard, the value is embedded via an `EXTERNAL` type. Other uses of the `EXTERNAL` type can be found in the definition of the X.400 e-mail content (see Section 7.2 on page 81) or the description of the nodes for FTAM file transfer (see on page 81) in particular.

Since 1994[2], the `EXTERNAL` type is formally defined with the structured type of Figure 14.1 on the next page.

This definition preserves the encoding compatibility with the type in the previous version (1990). Indeed, the structured type is used to denote the abstract value definition and clarify the components' combinations, but the encoding rules of the `EXTERNAL` will not be necessarily referencing the associated type.

The alternative `syntax` identifies both the abstract syntax and the transfer syntax (i.e. the encoding rules, see footnote 12 on page 15) with a single object identifier, whereas `presentation-context-id` identifies one of the presentation contexts that were negotiated. The alternative `context-negotiation` (consisting of the components `direct-reference` and `indirect-reference` of the 1990 version type) is used when the abstract syntax negotiation is taking place: the object identifier of the transfer syntax is then added to the presentation context identifier that is being negotiated. These last two alternatives apply only in an OSI environment[3].

Note that the only way to indicate the embedded value as an octet string actually groups together (in terms of encoding) the three alternatives `encoding` of the 1990 version.

Unfortunately, when the `EXTERNAL` type has been designed, all the Presentation layer principles were not fully grasped and the very notion of embedded value was a bit unusual. Because of mistakes in its design, the use of the `EXTERNAL` type has been strongly unrecommended since the 1994 edition of ASN.1. The latter proposes to replace it, among

[2]The definition that is given here can be found precisely in the ASN.1:1997 standard because the ASN.1:1994 standard was subject to some amendments on this point.

[3]As a result, these cannot be used in a switching environment [ETSI60], for example.

```
EXTERNAL ::= [UNIVERSAL 8] IMPLICIT SEQUENCE {
   identification  CHOICE {
      syntax[b]  OBJECT IDENTIFIER,
      presentation-context-id[c]  INTEGER,
      context-negotiation[d]  SEQUENCE {
         presentation-context-id  INTEGER,
         transfer-syntax  OBJECT IDENTIFIER } },
   data-value-descriptor  ObjectDescriptor OPTIONAL,
   data-value[g]  OCTET STRING }
```

[c]The footnotes *b*, *c* and *d* are the same as those in Figure 14.3 on page 305.

[g]The italic font denotes the words that have changed when comparing with the `SEQUENCE` type associated with the types `EMBEDDED PDV` (see Figure 14.3 on page 305) and `CHARACTER STRING` (see Figure 14.4 on page 307).

Figure 14.1: `SEQUENCE` type associated with the `EXTERNAL` type (defined in an automatic-tagging environment)

others, with the `EMBEDDED PDV` type, which is described in the next section, or with the `INSTANCE OF` type (see Section 15.9.1 on page 357).This updating, of course, does not ensure the encoding compatibility except if we use a type such as '`CHOICE { external EXTERNAL, embedded-pdv EMBEDDED PDV }`' in a BER encoding context.

14.1.2 Reference Manual

Type notation

ExternalType → `EXTERNAL`

⟨1⟩ A value of type `EXTERNAL` represents three items of information: the encoding of a single value (which is not necessarily of an ASN.1 type), an abstract syntax that contains this value and the transfer syntax (*i.e. the encoding rules*, see footnote 12 on page 15) used for this value.
⟨2⟩ This type has tag no. 8 of class `UNIVERSAL`. It has the same tag as the type `INSTANCE OF` (see on page 358).
⟨3⟩ The type `EXTERNAL` is defined (in an automatic-tagging environment) by the structured type given in Figure 14.1.
⟨4⟩ The type `EXTERNAL` can be subtyped according to its associated type defined in Figure 14.1, i.e. by a single value (see production *SingleValue* on page 261) and by constraint on its components (see production *InnerTypeConstraints* on page 277).

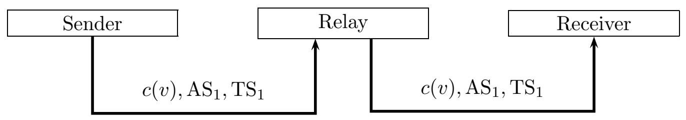

Sender
Relay
Receiver
$c(v), \mathrm{AS}_1, \mathrm{TS}_1$
$c(v), \mathrm{AS}_1, \mathrm{TS}_1$

Figure 14.2: Abstract and transfer syntax identifiers hand-over when relaying an embedded value

⟨5⟩ Instead of the `EXTERNAL` type, it is recommended to use the `INSTANCE OF` type, a subtyped `EXTERNAL`, the `EMBEDDED PDV` type or the type `CHOICE {external EXTERNAL, embedded-pdv EMBEDDED PDV}`.
⟨6⟩ If there is no negotiation on the presentation layer, it is still possible to use an `EXTERNAL` type as explained in [ISO8824-2, annex C].

Value notation

ExternalValue → *SequenceValue*

⟨7⟩ The *SequenceValue* must conform to the `SEQUENCE` type associated with the `EXTERNAL` type as described in Figure 14.1 on the page before.

14.2 The `EMBEDDED PDV` type

14.2.1 User's Guide

The type `EMBEDDED PDV` was introduced in 1994 to make up for historical mistakes of the `EXTERNAL` type. Unfortunately, instead of proposing a light version more specifically adapted for communicating applications based on message storing or relay systems, numerous alternatives were added to the `EXTERNAL` type in order to replace it.

If such applications store or relay embedded messages, the abstract syntax and transfer syntax object identifiers should be kept to forward these messages in the same format as that was received (see Figure 14.2). Unfortunately no alternative of the choice '`identification`' of the `EXTERNAL` type can declare two object identifiers and the alternative `presentation-context-id` that identifies the presentation context ⟨abstract syntax, transfer syntax⟩ is not appropriate since it does not guarantee that the presentation context identifier is the same for the reception connection and the re-emission connection. Ensuring that these two object identifiers are retained has motivated the addition of the alternative `syntaxes` to the type `EMBEDDED PDV`.

From the abstract syntax viewpoint, a value of type `EMBEDDED PDV` should therefore conform to the structured type described in Figure 14.3 on page 305.

The alternatives `syntax`, `presentation-context-id` and `context-negotiation` are the same as those of `EXTERNAL` type. The alternative `syntaxes` references the abstract and transfer syntaxes by their respective object identifier; this is the main difference with the type `EXTERNAL`.

The alternative `transfer-syntax` references only the transfer syntax (i.e. the encoding rules, see footnote 12 on page 15) by its object identifier. It proves particularly useful in case of data compression or encryption because it can state out clearly the method applied to the abstract syntax.

Finally, the alternative `fixed` is used if the abstract and transfer syntaxes resulted from a negotiation between the two applications; in this case the encoding is more compact[4] since transmitting the syntaxes' object identifiers with each embedded value becomes unnecessary. This alternative will be used in case the ASN.1 type of the embedded data is defined in another specification (this has obviously nothing to do with a type imported in the current specification).

For a more detailed description of the functionalities offered by the `EMBEDDED PDV`, the reader may refer to [Lar96].

All the alternatives and components of the `EMBEDDED PDV` type can be subtyped using the `WITH COMPONENTS` constraint discussed in Section 13.9 on page 277. In order to keep only the alternative `syntaxes` that constitutes the difference between the types `EMBEDDED PDV` and `EXTERNAL`, we write:

```
EMBEDDED PDV (WITH COMPONENTS {
    ...,
    identification (WITH COMPONENTS {syntaxes}) })
```

One can come across a particular use of the `OCTET STRING` type in some protocols (ETSI specifications or Internet RFC, for example) that are specified in ASN.1 and make use of another protocol also specified

[4]The first documents on the `EMBEDDED PDV` type gave two kinds of encoding (see footnote 13 on page 412): the complete encoding was used for the first transmission of an embedded value and featured the abstract syntax and transfer syntax object identifiers. The 'elementary' encoding was used for the following transmissions where the first octet stored the presentation context identifier. These two forms were eventually not standardized.

in ASN.1: the former collects the second protocol's PDV and embeds it in its own PDV as an octet string. Moreover the embedded PDV is not necessarily encoded with the same encoding rules as the embedding PDV.

To solve such a demand (that is already provided by some compilers), the ASN.1 working group has investigated the introduction of two new subtype constraints called `CONTAINING` and `ENCODED BY` that can be applied on the `OCTET STRING` type (see Section 13.10 on page 283) :

```
T ::= OCTET STRING ( CONTAINING EmbeddedType
                     ENCODED BY oid )
```

14.2.2 Reference Manual

Type notation

EmbeddedPDVType → `EMBEDDED PDV`

⟨1⟩ The `EMBEDDED PDV` type makes it possible to use a specific encoding for an abstract value while making the transfer syntax negotiation easier on the Presentation layer. A value of type `EMBEDDED PDV` represents the encoding of a single value (which is not necessarily an ASN.1 type), as well as an abstract syntax that contain these values and the transfer syntax (i.e. the encoding rules) used for separating this value from other values of the same class. The set of values of this type includes all the possible values of all the possible abstract syntaxes.
⟨2⟩ This type has tag no. 11 of class `UNIVERSAL`.
⟨3⟩ The type `EMBEDDED PDV` is defined (in an automatic-tagging environment) by the structured type given in Figure 14.3 on the next page.
⟨4⟩ This type can be subtyped according to its associated `SEQUENCE` type defined in Figure 14.3 on the next page, i.e. by a single value (see production *SingleValue* on page 261) and by constraint on its components (see production *InnerTypeConstraints* on page 277).

Value notation

EmbeddedPDVValue → *SequenceValue*

⟨5⟩ The value *SequenceValue* must conform to the `SEQUENCE` type associated with the `EMBEDDED PDV` defined in Figure 14.3 on the next page.

```
EMBEDDED PDV ::= [UNIVERSAL 11] IMPLICIT SEQUENCE {
   identification  CHOICE {
      syntaxes[a]  SEQUENCE {
         abstract  OBJECT IDENTIFIER,
         transfer  OBJECT IDENTIFIER },
      syntax[b]  OBJECT IDENTIFIER,
      presentation-context-id[c]  INTEGER,
      context-negotiation[d]  SEQUENCE {
         presentation-context-id  INTEGER,
         transfer-syntax  OBJECT IDENTIFIER },
      transfer-syntax[e]  OBJECT IDENTIFIER,
      fixed[f]  NULL },
   data-value[g]  OCTET STRING }
```

[a]Object identifiers of the abstract syntax and transfer syntax (encoding rules).

[b]Single object identifier for denoting the abstract syntax and encoding rules.

[c]Identifier of a negotiated presentation context (i.e. pair of abstract and transfer syntaxes, used only in an OSI environment).

[d]Used in a presentation context negotiation that may appear either at the beginning of an OSI connection or during this connection if there was a change of context (only in an OSI environment).

[e]In this case, the abstract syntax (for example, an indication that it is an ASN.1 type) is assumed fixed by the application designers and known by both the sender and the receiver.

[f]The abstract syntax and encoding rules are known by the sender and the receiver.

[g]The italic font denotes the words that have changed when comparing with the `SEQUENCE` type associated with the types `EXTERNAL` (see Figure 14.1 on page 301) and `CHARACTER STRING` (see Figure 14.4 on page 307).

Figure 14.3: `SEQUENCE` type associated with the `EMBEDDED PDV` type (defined in an automatic-tagging environment)

14.3 The `CHARACTER STRING` type

14.3.1 User's Guide

The `CHARACTER STRING` type is the concrete application of the `EMBEDDED PDV` type to the special case of a character string. It prevents from restricting a specification to one of the character string types of Chapter 11 and to their standardized encoding (BER or PER). The *character abstract syntax* and the *character transfer syntax*[5] can be negotiated using the Presentation layer's standard mechanism as in Figure 3.2 on page 22 (for cases where the transfer is unrelayed) or declared with the embedded character string if the environment is appropriate (in this case, the sender should consult a directory service like the one presented in Section 7.3 on page 83 to know the transfer and abstract syntaxes supported by the receiver).

A character abstract or transfer syntax can be defined by any organization entitled to allocate object identifiers in the world-wide registration tree (see Section 10.8 on page 153). As described on page 185, the standard [ISO10646-1] defines an abstract syntax (and consequently an object identifier) for every one of its subsets of characters and for all combinations of these subsets. It defines also a few character transfer syntaxes.

A recently standardized alphabet only needs to be allocated an object identifier to be used in an ASN.1 specification by means of the `CHARACTER STRING` type. To prevent from associating two object identifiers to a character string, these can be indicated in a `WITH COMPONENTS` constraint when they are constant:

```
My-string ::= CHARACTER STRING (WITH COMPONENTS {
    ...,
    identification (WITH COMPONENTS {
        syntaxes ({abstract my-OID-AS,
                   transfer my-OID-TS}) })})
```

The `SEQUENCE` type that is used for defining abstract values of type `CHARACTER STRING` is equivalent to the `EMBEDDED PDV` type's; it is described in Figure 14.4 on the next page.

[5]A character abstract syntax is an abstract syntax that accepts only character strings as values. An example of character transfer syntax is given in Table 11.3 on page 191.

```
CHARACTER STRING ::= [UNIVERSAL 29] SEQUENCE {
   identification  CHOICE {
      syntaxes[a]  SEQUENCE {
         abstract  OBJECT IDENTIFIER,
         transfer  OBJECT IDENTIFIER },
      syntax[b]  OBJECT IDENTIFIER,
      presentation-context-id[c]  INTEGER,
      context-negotiation[d]  SEQUENCE {
         presentation-context-id  INTEGER,
         transfer-syntax  OBJECT IDENTIFIER },
      transfer-syntax[e]  OBJECT IDENTIFIER,
      fixed[f]  NULL },
   string-value[g]  OCTET STRING }
```

[a]The footnotes *a* to *f* are the same as those in Figure 14.3 on page 305.

[g]The italic font denotes the words that have changed when comparing with the SEQUENCE type associated with the EXTERNAL types (see Figure 14.1 on page 301) and EMBEDDED PDV (see Figure 14.3 on page 305).

Figure 14.4: SEQUENCE type associated with the CHARACTER STRING type (defined in an automatic tagging environment)

14.3.2 Reference Manual

Type notation

UnrestrictedCharacterStringType → CHARACTER STRING

⟨1⟩ The possible values of the CHARACTER STRING type are all the possible strings of all the possible character abstract syntaxes. The abstract syntax may belong to the presentation context set allocated in an instance of communication or can be directly referenced when using the CHARACTER STRING type.

⟨2⟩ This type has tag no. 29 of class UNIVERSAL.

⟨3⟩ The type CHARACTER STRING is defined (in an automatic tagging environment) by the structured type given in Figure 14.4.

⟨4⟩ This type can be subtyped according to its associated SEQUENCE type (see Figure 14.4), i.e. by a single value (production *SingleValue* on page 261) and by constraint on its components (production *InnerTypeConstraints* on page 277). The *PermittedAlphabet* constraint (see on page 269) is not allowed. The standard also allows a size constraint (production *SizeConstraint* on page 267) but does not mention the measure; it is recommended not to use this constraint but wait for a technical corrigendum that will remove this ambiguity (see also next rule).

⟨5⟩ When applied to the `CHARACTER STRING` type, the size constraint (production *SizeConstraint* on page 267) must limit the number of characters (not of octets) of the embedded strings. But the definition of the 'character' notion depends on the character abstract syntax. Should these ambiguities be removed, limiting the number of octets in the embedded strings remains possible with a constraint of the form `CHARACTER STRING (WITH COMPONENTS {..., string-value (SIZE (1..50) -- in octets --)})`.

Value notation

UnrestrictedCharacterStringValue → *SequenceValue*

⟨6⟩ The value *SequenceValue* must conform to the associated `SEQUENCE` type defined in Figure 14.4 on the page before.

Chapter 15

Information object classes, objects and object sets

Contents

There is a further difficulty that deserves attention. Wise men, if they try to speak their language to the common herd instead of its own, cannot possibly make themselves understood. There are a thousand kinds of ideas which it is impossible to translate into popular language. Conceptions that are too general and objects that are too remote are equally out of its range [...].

Jean-Jacques Rousseau, *The Social Contract*.

In ASN.1, the concept of information object class is used to formally represent properties uncovered by the notions of type and value in particular. These properties can very often be interpreted as semantic links between types and values (among others). They allow the specifier to leave open areas (gaps) in the specification while restricting how these gaps should be filled. Even if the information objects are never encoded, they are nonetheless used by ASN.1 compilers when generating the encoders and decoders.

The reader who may get confused between the information object classes and the constructed types (or the information objects and the abstract values) should think of the former as being of a higher conceptual level; in other words, types and abstract values can be defined without using information object classes but the definition of an information object class must rely on types or values and even sometimes on other information object classes, objects or object sets.

15.1 Introduction to information object classes

An information object class is, first of all, a class, in other words "*a group of people or things sharing common characteristics*" (*The Collins Pocket*). In ASN.1, these characteristics can reference a type, a value, a value set, an information object or an information object set.

Imagine a class of computing functions or operations with four *fields*[1], given in Figure 15.1 on the next page: for every function, the

[1]The term 'attribute' is more common in the literature on object-oriented design, but ASN.1 classes do not fit in this category.

Operation name

ASN.1 type of the argument:	_____________
ASN.1 type of the result	: _____________ (`NULL` by default)
Error message list	: { _____, _____, _____ ... }
Identification code	: ____

Figure 15.1: Description form of a remote application

first field contains the argument type, the second one the result type, the third represents the set of error messages that may be returned and the fourth an identification code.

To carry out a remote execution of one of these functions, a communicating application should send a sequence of two components[2]: the identification code (to specify the function to be executed by the remote application) and a value that conforms to the ASN.1 type of the argument. In return, the application receives a value that conforms to the result type if the function execution was successful, and one of the specified error messages otherwise. We shall see that such a functioning is very close to the Remote Operation Service Element ROSE [ISO13712-1] and its information object class called `OPERATION`.

The information object classes appeared in the 1994 edition of the ASN.1 standard [ISO8824-2]. They take up, in principle at least, the notion of macros (see Chapter 16) and the `ANY` type with its `DEFINED BY` clause (see Section 12.8 on page 241), which are no longer part of the standard. They are used to 'open' a specification in order to offer degrees of freedom to its future users to let them define information objects specific to their application domains.

Contrary to what their names may suggest, the classes and information objects bear no relation to the object paradigm. In particular, they support no inheritance mechanism. We should not confuse, however, the notion of class with that of type nor should we confuse those of object and value: objects are never encoded to be transmitted, but classes are nonetheless taken into account by ASN.1 compilers by means of dedicated type declarations and subtype constraints presented in Section 15.7 on page 341. Besides, we shall see that they authorize a certain

[2]Note that they are two actual components of a `SEQUENCE` type, and not two fields of the class (which cannot be encoded).

degree of freedom to communicating applications and thereby make encoders and decoders more flexible.

Although a type needs only to be defined to implicitly contain values (remember an abstract type is semantically equivalent to a value set), a class would be better interpreted as a form (see the one of Figure 15.1 on the page before) which should be filled each time a new object is created. The objects thus created can be gathered in an *object set*.

Classes can be given a *user-friendly* syntax to allow the designer of the communicating application to define the properties of information objects in a layout very close to a form. The information object classes can undoubtedly draw an intuitive link between the ASN.1 specification of the data transfer strictly speaking and specific needs that may arise for many communicating applications.

A description similar to the one given in this chapter can be found in a more condensed way in [Mit94].

15.2 Default syntax of information objects and classes

15.2.1 User's Guide

Syntactically (only), a class definition can be compared to an imaginary constructed type whose keyword would be `CLASS`. The class name is spelt in capital letters[3] and every one of its fields is denoted by an identifier beginning with an ampersand "`&`" followed by a lower-case or an upper-case letter as we shall see later on. The "`&`" sign makes a distinction between the *fields* of a class and the components of a `SEQUENCE` or `SET` type.

The field identifiers are sometimes followed by a type, a reference to another class or another field of the same class; they can be marked `OPTIONAL`, `DEFAULT` or `UNIQUE`. But contrary to structured types, the class fields do not only contain values, they can also contain a type, a value set, an information object or an information object set; hence the markers `OPTIONAL` and `DEFAULT` have been generalized from the `SEQUENCE` and `SET` types to the information object classes.

[3]Historically, the macro names (see Chapter 16) included only upper-case letters and information object classes, which are meant to replace them, follow the same rule.

A function of any computing language can be modeled by the following information object class, which is just another way of representing the form drawn in Figure 15.1 on page 311:

```
FUNCTION ::= CLASS {
  &ArgumentType ,
  &ResultType  DEFAULT NULL,
  &Errors      ERROR OPTIONAL,
  &code        INTEGER UNIQUE }
```

Any function is, therefore, modeled by an information object that is identified by a distinct code[4] and includes the following information: the argument type, the result type (by default, the `NULL` type) and optionally an error list, every one of these errors being modeled by an object of another class called `ERROR`.

For a function that adds two integers, we define the following information object with a name beginning with a small letter (like an abstract value) and the necessary reference to the form (i.e. the information object class) to be filled to define this object:

```
addition-of-2-integers FUNCTION ::= {
  &ArgumentType SEQUENCE { a INTEGER, b INTEGER },
  &ResultType   INTEGER,
  -- empty error list by default
  &code         1 }
```

Such a mixture of heterogeneous information (including two types, a value and an object set) can obviously not be represented only by ASN.1 types and values.

As clearly shown in the previous example, depending on whether a field name begins with a lower-case or an upper-case letter and is followed by a word beginning with a small or a capital letter, the field belongs to a certain category. In fact, there are seven different categories of fields as shown in Table 15.1 on the following page. If the field name begins with a lower-case letter, it refers to a value or an information object; if it begins with an upper-case letter, it refers to a type, a value set or an information object set.

[4]The scope within which the requirement on distinct codes must be respected is introduced later (see rule ⟨7⟩ on page 318).

If the field name starts with	**and if it is followed by**	**then the field of the object contains**
`&Upper-case`	nothing	a type
`&lower-case`	a type or a type reference (`Upper-case`)	a fixed-type value
`&lower-case`	a type field (`&Upper-case`)	a variable-type value
`&Upper-case`	a type or a type reference (`Upper-case`)	a fixed-type value set
`&Upper-case`	a type field (`&Upper-case`)	a variable-type value set
`&lower-case`	a class name (`UPPER-CASES`)	an information object
`&Upper-case`	a class name (`UPPER-CASES`)	an information object set

Table 15.1: The seven categories of fields of an information object class

We now define a new information object class for our functions that includes (at least) one field of each of the seven field categories mentioned:

```
OTHER-FUNCTION ::= CLASS {
  &code                INTEGER (0..MAX) UNIQUE,
  &Alphabet            BMPString
    DEFAULT {Latin1 INTERSECTION Level1},
  &ArgumentType        ,
  &SupportedArguments  &ArgumentType OPTIONAL,
  &ResultType          DEFAULT NULL,
  &result-if-error     &ResultType DEFAULT NULL,
  &associated-function OTHER-FUNCTION OPTIONAL,
  &Errors              ERROR DEFAULT
    {rejected-argument | memory-fault} }

rejected-argument ERROR ::=
  {-- object definition --}
memory-fault ERROR ::=
  {-- object definition --}
```

The information object describing the addition of two integers can then be written as follows (we do not quite respect the class field ordering, which is perfectly allowed):

```
other-addition-of-2-integers OTHER-FUNCTION ::= {
  &ArgumentType        Pair,
  &SupportedArguments {PosPair | NegPair},
  &ResultType          INTEGER,
  &result-if-error     0,
  &code                1 }
Pair ::= SEQUENCE {a INTEGER, b INTEGER}
PosPair ::= Pair (WITH COMPONENTS {a(0..MAX), b(0..MAX)})
NegPair ::= Pair (WITH COMPONENTS {a(MIN..0), b(MIN..0)})
```

We now give the meaning of each field of the class `OTHER-FUNCTION`. The field name `&code` begins with a lower-case letter and is followed by an ASN.1 type. `&code` is a *fixed-type value field*, which means that, within an information object, this field contains an ASN.1 value that conforms to the specified type. Besides, the `UNIQUE` marker ensures that for every object set of class `OTHER-FUNCTION`, two objects cannot have the same identification code. Only a fixed-type value field can be an *identifier field* (whose type is generally `INTEGER`, `OBJECT IDENTIFIER` or a `CHOICE` between these two types[5]).

The field name `&Alphabet` starts with a capital letter and is followed by an ASN.1 type. `&Alphabet` is a *fixed-type value set field*, which means that, within an object, this field contains a value set of the specified type. It is, for example, the alphabet that must be respected by all the messages a function may have to display. The notion of value set (presented in Section 15.5 on page 329) provides the flexibility offered by the set operators of Section 13.11 on page 285 (and particularly of Figure 13.3 on page 286), as illustrated by the default value of the `&Alphabet` field in the `OTHER-FUNCTION` class definition: it is the character string set of the `Latin1` alphabet on the implementation level 1 of the `BMPString` type (see Section 11.11 on page 189).

The field name `&ArgumentType` begins with an upper-case letter, and is followed by neither a type nor a class. It is a *type field*, which means that in an object, the field contains an ASN.1 type. It enables the user to indicate the argument type, that differs for each function. We see further on that such a field can be used to specify an *open type*, equivalent to the obsolete `ANY` type which has been excluded of the standard since

[5]We made a similar remark on page 242 for the ASN.1:1990 `ANY DEFINED BY` type.

1994 (see Section 12.8 on page 241). Similarly, `&ResultType` is a type field. If it is not defined in the object, it is considered to have the `NULL` *type* (and not 'value') by default.

The field name `&result-if-error` begins with a lower-case letter; it is followed by the name of a type field. `&result-if-error` is a *variable-type value field*. The term 'variable' means that the type is not the same for all objects, i.e. it is not fixed in the class definition. The result returned if an error occurs when running the function must be of the same type as all the other potential results. For this reason, the default value (the `NULL` value) of the `&result-of-error` field must conform to the default type of the `&ResultType` field (the `NULL` type).

The field name `&SupportedArguments` begins with an upper-case letter, and is followed by the name of a type field. `&SupportedArguments` is a *variable-type value set field*, which means that, in an object, this field contains a value set that conforms to the type indicated in the field `&ArgumentType` (that is the smallest set of values to be taken into account in all the implementations of this function without generating the error message `rejected-argument`). For the `other-addition-of-2-integers` object, the supported arguments are pairs whose elements are either both positive or both negative (why not?!).

The field name `&associated-function` begins with a lower-case letter, and is followed by the name of an information object class. It is an *object field*[6], which means that, in an object, this field contains a reference to another object. This is used to 'factorize' a collection of information shared by several objects (in the present case, the same function, featuring information that should be shared by others, may be associated with several other functions). In the same way as ASN.1 types can be recursive (or self-referencial) an information object class can be referenced in its own definition. It is not allowed, however, to define recursive objects (as it is forbidden to define recursive values). The field `&associated-function` is marked `OPTIONAL` to indicate that a function execution does not systematically require the prior execution of an associated function.

The field name `&Errors` begins with an upper-case letter, and is followed by the name of an information object class. `&Errors` is an *object set*

[6]Note that an 'object field' is a 'field of an object', but all the fields of an object are not necessarily 'object fields'. In the rest of this text, the expression 'object field' will always reference one of the seven categories of fields of an object (see Figure 15.1 on page 314).

field, which means that, in an object, this field contains a set of objects that are instances of the given class. An object set is denoted similarly as a value set, usually separating the object references by a vertical bar "`|`", and delimited by curly brackets as shown in the default value set `{rejected-argument|memory-fault}` associated with the field `&Errors` in the definition of the class `OTHER-FUNCTION` on page 314.

As far as the semantic model of ASN.1 is concerned (see Section 9.4 on page 121), even if two information object classes have syntactically identical definitions but for the name, they are considered as different and no compatibility exists between them, i.e. an object of one class cannot be used to define an object of the other class:

```
FUNCTION-BIS ::= OTHER-FUNCTION
add FUNCTION-BIS ::=
  other-addition-of-2-integers -- forbidden
```

15.2.2 Reference Manual

Class notation

ObjectClass → *DefinedObjectClass*
| *ObjectClassDefn*
| *ParameterizedObjectClass*

ObjectClassDefn → `CLASS` "{" *FieldSpec* "," ...$^+$ "}"
WithSyntaxSpec

FieldSpec → *TypeFieldSpec*
| *FixedTypeValueFieldSpec*
| *VariableTypeValueFieldSpec*
| *FixedTypeValueSetFieldSpec*
| *VariableTypeValueSetFieldSpec*
| *ObjectFieldSpec*
| *ObjectSetFieldSpec*

⟨1⟩ The name of all the fields specified in a class definition must be distinct except if they differ by their (upper/lower) case.

⟨2⟩ We call:

– *type field*, a field (of an information object) that references a type specified in ASN.1;

– *fixed-type value field*, a field that references a value whose type is specified at the right-hand side of this field's name in the information object class definition;

– *variable-type value field*, a field that references a value whose type is unknown when defining the information object class;

– *fixed-type value set field*, a field that references a value set whose type is specified on the right-hand side of this field's name in the information object class definition;

– *variable-type value set field*, a field that references a value set whose type is unknown when defining the information object class;

– *object field* (see footnote 6 on page 316), a field that references another information object (not necessarily of the same class);

– *object set field*, a field that stores an information object set.

TypeFieldSpec → typefieldreference *TypeOptionalitySpec*

TypeOptionalitySpec → `OPTIONAL`
| `DEFAULT` *Type*
| ϵ

⟨3⟩ *TypeFieldSpec* is a type field, which means that in an information object definition this field stores any type that can be specified in ASN.1.
⟨4⟩ If the `OPTIONAL` clause is used, the field needs not be allocated within the object.
⟨5⟩ If the `DEFAULT` clause is used, *Type* indicates the default type of the type field if this field is not defined within the object.

FixedTypeValueFieldSpec → valuefieldreference *Type* *Unique* *ValueOptionalitySpec*

⟨6⟩ *FixedTypeValueFieldSpec* is a fixed-type value field, which means that in an information object definition this field stores a value of type *Type*.

Unique → `UNIQUE`
| ϵ

⟨7⟩ The value of an identifier field (i.e. followed by the `UNIQUE` keyword) must be unique within any information object set of this class (see rule ⟨8⟩ on page 332).
⟨8⟩ If the keyword `UNIQUE` is used, the `DEFAULT` clause cannot be used.
⟨9⟩ In the case of a dynamically extensible object set (see on page 331) whose objects have a field marked `UNIQUE`, the index of an object that has been removed may be re-used later on for another object. Checking this property is down to the application. This problem may occur for an

index of type `INTEGER`, for instance, but cannot be if the index is typed `OBJECT IDENTIFIER`.

ValueOptionalitySpec → `OPTIONAL`
| `DEFAULT` *Value*
| ϵ

⟨10⟩ If the `OPTIONAL` clause is used, the field needs not be allocated within the object.
⟨11⟩ If the `DEFAULT` clause is used, *Value* indicates the default value of the value field if it is not defined within the object.
⟨12⟩ *Value* should be of the *Type* appearing in *FixedTypeValueFieldSpec* or of a type that is compatible with *Type* according to the semantic model of ASN.1 (see Section 9.4 on page 121).

VariableTypeValueFieldSpec → valuefieldreference *FieldName* *ValueOptionalitySpec*

⟨13⟩ *VariableTypeValueFieldSpec* is a value field whose type is not known when writing the specification; in an information object definition, this field stores a value whose type is specified in the same or some other information object.
⟨14⟩ *FieldName* must be the name of a type field (production *TypeFieldSpec* on the preceding page) that is not necessarily in the same information object class.
⟨15⟩ If the `OPTIONAL` alternative is used for *ValueOptionalitySpec*, the field denoted by *FieldName* must be marked `OPTIONAL` or `DEFAULT` in the information object class where it is defined.
⟨16⟩ If the '`DEFAULT` *Value*' alternative is used in *ValueOptionalitySpec*, the field denoted by *FieldName* must include the '`DEFAULT` *Type*' clause in the information object class where it is defined, and *Value* must be either of type *Type* or of a type that is compatible with *Type* according to the semantic model of ASN.1 (see Section 9.4 on page 121).

FixedTypeValueSetFieldSpec → valuesetfieldreference *Type* *ValueSetOptionalitySpec*

⟨17⟩ *FixedTypeValueSetFieldSpec* is a fixed-type value set field, which means that in an information object definition, this field stores a value set of type *Type*.

ValueSetOptionalitySpec → `OPTIONAL`
| `DEFAULT` *ValueSet*
| ϵ

⟨18⟩ If the `OPTIONAL` clause is used, the field needs not to be allocated within the object.
⟨19⟩ If the clause `DEFAULT` is used, *ValueSet* indicates the default value set of the value set field if it is not defined in the object.
⟨20⟩ *ValueSet* must be a set of values of type *Type* or of a type that is compatible with *Type* according to the semantic model of ASN.1 (see Section 9.4 on page 121).

VariableTypeValueSetFieldSpec →
valuesetfieldreference *FieldName* *ValueSetOptionalitySpec*

⟨21⟩ *VariableTypeValueSetFieldSpec* is a value set field whose type is not known when writing the specification; this field stores a set of values that are all of a type specified in the same or some other information object.
⟨22⟩ *FieldName* is the name of a type field (production *TypeFieldSpec* on page 318) that is not necessarily in the same information object class.
⟨23⟩ If the `OPTIONAL` alternative of *ValueSetOptionalitySpec* is allowed, the field denoted by *FieldName* must be marked `OPTIONAL` or `DEFAULT` in the information object class where it is defined.
⟨24⟩ If the alternative '`DEFAULT` *ValueSet*' is used for *ValueSetOptionalitySpec*, then the field denoted by *FieldName* must include the '`DEFAULT` *Type*' clause in the information object class where it is defined, and *ValueSet* must be of type *Type* or of a type that is compatible with *Type* according to the semantic model of ASN.1 (see Section 9.4 on page 121).

ObjectFieldSpec → objectfieldreference *DefinedObjectClass*
ObjectOptionalitySpec

⟨25⟩ *ObjectFieldSpec* is an object field, which means that in an information object definition, this field stores another information object (not necessarily of the same class).

ObjectOptionalitySpec → `OPTIONAL`
| `DEFAULT` *Object*
| ϵ

⟨26⟩ If the `OPTIONAL` clause is used, the field needs not be allocated within the object.
⟨27⟩ If the `DEFAULT` clause is used, *Object* indicates the default object for the object field if it is not defined within the object.

⟨28⟩ *Object* must be an object of the class referenced by *DefinedObjectClass*.

ObjectSetFieldSpec → objectsetfieldreference *DefinedObjectClass* *ObjectSetOptionalitySpec*

⟨29⟩ *ObjectSetFieldSpec* is an information object set, which means that within an information object definition, this field stores an information object set (not necessarily of the same class).

ObjectSetOptionalitySpec → `OPTIONAL`
| `DEFAULT` *ObjectSet*
| ϵ

⟨30⟩ If the `OPTIONAL` clause is used, the field needs not to be allocated within the object.
⟨31⟩ If the `DEFAULT` clause is used, *ObjectSet* indicates the default object set for the object set field if it is not defined within the object.
⟨32⟩ All the objects in *ObjectSet* must be of the class referenced by *DefinedObjectClass*.

FieldName → *PrimitiveFieldName* "." $\cdots^{+}$

⟨33⟩ If there exists such a series of fields that: the first field is in a class, say `C`; every one of the other fields belongs to the class referenced by the previous field; and the last field belongs to `C`; then one of the fields at least must be marked `OPTIONAL` or `DEFAULT`. This rule prevents recursive definitions of classes where none of the objects has a finite representation.

PrimitiveFieldName → typefieldreference
| valuefieldreference | valuesetfieldreference
| objectfieldreference | objectsetfieldreference

Object notation

Object → *ObjectDefn* | *DefinedObject*
| *ObjectFromObject* | *ParameterizedObject*

⟨34⟩ An object definition cannot be recursive even if the associated class definition is recursive (see rule ⟨33⟩ above).

ObjectDefn → *DefaultSyntax*
| *DefinedSyntax*

⟨35⟩ The production *DefaultSyntax* can only be used if the class definition has no associated user-friendly syntax introduced by the keywords

`WITH SYNTAX`. If such a syntax *WithSyntaxSpec* is associated with the class definition, the *DefinedSyntax* production (see on page 326) must be used.

DefaultSyntax → "{" *FieldSetting* "," ···* "}"

FieldSetting → *PrimitiveFieldName Setting*

⟨36⟩ There must be exactly one *FieldSetting* in the object for every *FieldSpec* not marked `OPTIONAL` or `DEFAULT` in the class definition (see production *ObjectClassDefn* on page 317) and at most one *FieldSetting* otherwise.
⟨37⟩ The *FieldSetting*s may appear in any order in the object.

Setting → *Type* | *Value*
| *ValueSet* | *Object*
| *ObjectSet*

⟨38⟩ The alternative of *Setting* must conform to the corresponding field of the class (the *Type* alternative if it is a type field, the *Value* alternative if it is a fixed-type or variable-type value, and so on).
⟨39⟩ For a fixed-type value field (respectively fixed-type value set field), *Setting* is a value (respectively value set) of the type specified in the field or of a type that is compatible with it according to the semantic model of ASN.1 (see Section 9.4 on page 121)
⟨40⟩ For a variable-type value field, *Setting* is a value of the type specified by the *Setting* corresponding to the type field indicated in the right-hand part of the variable-type value field or of a type that is compatible with it according to the semantic model of ASN.1 (see Section 9.4 on page 121). The *Value* production, therefore, must be directly used instead of the production *OpenTypeFieldVal* on page 348 (in which the value is preceded by its type and the symbol ":") since the type is not open but fixed by another field of the information object.
⟨41⟩ For a variable-type value set field, *Setting* is a value set of the type specified by the *Setting* corresponding to the type field indicated in the right-hand part of the variable-type value set field or of a type that is compatible with it according to the semantic model of ASN.1 (see Section 9.4 on page 121).
⟨42⟩ For an object field (respectively an object set field), *Setting* is an object (respectively an object set) of the class specified in the field.

15.3 User-friendly syntax

15.3.1 User's Guide

We proceed our journey through the land of information objects and classes showing how we can associate a specific syntax with a class definition to make specifications easier to design. Indeed, these are very often specified by communicating application designers or by other standardization groups who know their application domain but cannot be asked to have a thorough understanding of ASN.1 syntax (for the X.400 e-mail or X.500 directory standards for example).

We will see in Chapter 16 that this specific syntax, called *user-friendly* in the rest of the text, shares many common points but for a few details, with the macro notation which has no longer been part of the standard since 1994.

A user-friendly syntax, introduced by the keywords `WITH SYNTAX`, is denoted after a class definition. In the second place, we specify in curly brackets a kind of 'phrase with gaps'[7] whose words are all in capital letters and whose gaps are the class field names (i.e. they begin with the ampersand "`&`"). Moreover, we may (it is not compulsory) use commas to distinguish the different parts of the phrase and thereby improve the clarity of the syntax. When a field is marked `OPTIONAL` or `DEFAULT` in the class definition, the corresponding part of the phrase must be written in square brackets to indicate that it is optional.

For the class `OTHER-FUNCTION` in the previous section, a user-friendly syntax could be:

```
OTHER-FUNCTION ::= CLASS {
  &code                 INTEGER (0..MAX) UNIQUE,
  &Alphabet             BMPString DEFAULT
     {Latin1 INTERSECTION Level1},
  &ArgumentType         ,
  &SupportedArguments   &ArgumentType OPTIONAL,
  &ResultType           DEFAULT NULL,
  &result-if-error      &ResultType DEFAULT NULL,
  &associated-function  OTHER-FUNCTION OPTIONAL,
  &Errors               ERROR DEFAULT
     {rejected-argument|memory-fault} }
```

[7]This phrase is no more than a formal representation of the form in Figure 15.1 on page 311.

```
WITH SYNTAX {
  ARGUMENT TYPE &ArgumentType,
  [SUPPORTED ARGUMENTS &SupportedArguments,]
  [RESULT TYPE &ResultType,
   [RETURNS &result-if-error IN CASE OF ERROR,]]
  [ERRORS &Errors,]
  [MESSAGE ALPHABET &Alphabet,]
  [ASSOCIATED FUNCTION &associated-function,]
  CODE &code }
memory-fault ERROR ::= {-- object definition --}
```

This user-friendly syntax definition calls for a few remarks:

- the words entirely in capital letters cannot be type or value keywords to prevent confusion (the forbidden keywords are listed in rule ⟨9⟩ on page 326);
- if successive optional parts appear, they cannot start with the same word;
- as the optional part '`RETURNS`' is inside the '`RESULT TYPE`' part, it is impossible to allocate a value to the `&result-if-error` field without valuating the `&ResultType` field;
- it is not strictly (syntactically) necessary to precede a field name by a word in capital letters but it is highly recommended for reading's sake.

When a user-friendly syntax is associated with a class definition, it should necessarily be used for defining objects of this class. An object is still defined in curly brackets but instead of the usual 'two-column' association of a 'value' to a field, we simply fill in the gaps (beginning with an ampersand "`&`") of the form. For the addition function of two integers introduced in the previous section, it gives:

```
addition-of-2-integers OTHER-FUNCTION ::= {
  ARGUMENT TYPE Pair,
  SUPPORTED ARGUMENTS {PosPair | NegPair},
  RESULT TYPE INTEGER,
  RETURNS 0 IN CASE OF ERROR,
  CODE 1 }
```

When we use the user-friendly syntax, words and commas must appear in the same order as in the class definition, whereas if an object is defined with the class default syntax, the field may appear in any order.

Finally, the user-friendly syntax should be defined with great care to make defining information easier without being too verbose. It should also conform to the semantic rules stated in the Reference Manual below (for this last point, the check-up can be performed by an ASN.1 compiler).

15.3.2 Reference Manual

User-friendly notation for an information object class

$$\begin{aligned} \textit{WithSyntaxSpec} \rightarrow\ & \texttt{WITH SYNTAX}\ \textit{SyntaxList} \\ \mid\ & \epsilon \end{aligned}$$

⟨1⟩ The `WITH SYNTAX` clause enables specifying a user-friendly syntax to make object definition easier.

$$\begin{aligned} \textit{SyntaxList} \rightarrow\ & \text{“\{”}\ \textit{TokenOrGroupSpec}\ \cdots^{+}\ \text{“\}”} \\ \textit{TokenOrGroupSpec} \rightarrow\ & \textit{RequiredToken} \\ \mid\ & \textit{OptionalGroup} \\ \textit{RequiredToken} \rightarrow\ & \textit{Literal} \\ \mid\ & \textit{PrimitiveFieldName} \end{aligned}$$

⟨2⟩ Every *PrimitiveFieldName* of the class must appear exactly once in the ‘phrase with gaps’ *SyntaxList*.

$$\textit{OptionalGroup} \rightarrow \text{“}\texttt{[}\text{”}\ \textit{TokenOrGroupSpec}\ \cdots^{+}\ \text{“}\texttt{]}\text{”}$$

⟨3⟩ If, during the parsing stage of the *OptionalGroup*, the next lexeme[8] of the stream is acceptable as the first lexeme of this *OptionalGroup*, the group is considered as ‘present’ (and all the lexical tokens that constitute the group must be found in the same order after the first lexeme), otherwise the group is considered as ‘absent’.
⟨4⟩ It is recommended that the first lexical token of an *OptionalGroup* should be a *Literal* (all in upper-case letters).
⟨5⟩ Every *OptionalGroup* must contain at least a *PrimitiveFieldName* or another *OptionalGroup* (which should conform to this rule recursively) to avoid collecting information that could not be stored in the object.
⟨6⟩ An *OptionalGroup* must be associated only with fields marked `OPTIONAL` or `DEFAULT` in the class.
⟨7⟩ The *OptionalGroup* must be defined so that no valuation of *Setting*

[8] The notions of parsing, lexeme and lexical token, which are necessary for understanding these rules are defined in Section 8.2 on page 98 and in Chapter 22 on page 463.

(see on the current page) can apply to several *FieldNames*. For example, the *SyntaxList* '`[LITERAL [A &field1] [B &field2]]`' is not ambiguous since the internal *OptionalGroups* do not begin with the same word.
⟨8⟩ If an *OptionalGroup* begins with a *Literal*, the lexical token following the *OptionalGroup* must also be a *Literal* different from every *Literal* that starts the immediately preceding *OptionalGroups* (this recalls the rule ⟨14⟩ on page 224 about requirements on distinct tags in groups of successive optional components of a `SEQUENCE` type).

$$\begin{aligned} \textit{Literal} &\rightarrow \textsf{word} \\ &\mid \text{“,”} \end{aligned}$$

⟨9⟩ word cannot be one of the following keywords: `BIT`, `BOOLEAN`, `CHARACTER`, `CHOICE`, `EMBEDDED`, `END`, `ENUMERATED`, `EXTERNAL`, `FALSE`, `INSTANCE`, `INTEGER`, `INTERSECTION`, `MINUS-INFINITY`, `NULL`, `OBJECT`, `OCTET`, `PLUS-INFINITY`, `REAL`, `RELATIVE-OID`, `SEQUENCE`, `SET`, `TRUE`, `UNION`. Since word consists only of upper-case letters, it cannot be confused with one of the keywords for character string types (see Chapter 11).
⟨10⟩ Commas are the only punctuation mark allowed in the user-friendly syntax.

User-friendly notation for an information object

$$\begin{aligned} \textit{DefinedSyntax} &\rightarrow \text{“\{”}\ \textit{DefinedSyntaxToken}\ \cdots^{*}\ \text{“\}”} \\ \textit{DefinedSyntaxToken} &\rightarrow \textit{Literal} \\ &\mid \textit{Setting} \end{aligned}$$

$$\begin{aligned} \textit{Literal} &\rightarrow \textsf{word} \\ &\mid \text{“,”} \end{aligned}$$

$$\begin{aligned} \textit{Setting} \rightarrow\ & \underline{\textit{Type}} \mid \underline{\textit{Value}} \\ \mid\ & \underline{\textit{ValueSet}} \mid \underline{\textit{Object}} \\ \mid\ & \underline{\textit{ObjectSet}} \end{aligned}$$

⟨11⟩ The rules ⟨38⟩ to ⟨42⟩ on page 322 apply also when *Setting* is used in a user-friendly syntax.
⟨12⟩ When *Literal*, spelt in upper-case letters (appearing in the object's *DefinedSyntax* at the same place as in the class's *SyntaxList*, see on the preceding page) can also be interpreted as a typereference or an objectsetreference, it must be interpreted as a word.
⟨13⟩ The *DefinedSyntax* is invalid if it does not allocate all the mandatory fields of the *SyntaxList* (see on the page before) of the class.

⟨14⟩ When a *Literal* of the *DefinedSyntax* appears in an *OptionalGroup* of the class's *WithSyntaxSpec*, the whole *OptionalGroup* is should be present in the object definition and one of the *PrimitiveFieldName*s of this *OptionalGroup* must be allocated a value through a *Setting* in this *DefinedSyntax*.

15.4 Example: the classes `ATTRIBUTE` and `MATCHING-RULE` of the X.500 recommendation

In Section 7.3 on page 83, we presented the X.500 directory service. It is a distributed information database that plays a major role in open systems inter-connection because it improves communication between applications. To introduce the description of an application (or any other communicating entity) in the directory database, the appropriate values are allocated to the corresponding attributes; to consult the database, a certain number of comparison rules are provided with arguments to obtain the description of the application(s) whose attributes satisfy the criterions.

The attributes (or 'attribute types') are information objects of the `ATTRIBUTE` class defined by[9]:

```
ATTRIBUTE ::= CLASS {
  &derivation             ATTRIBUTE OPTIONAL,
  &Type                   OPTIONAL,
  &equality-match         MATCHING-RULE OPTIONAL,
  &ordering-match         MATCHING-RULE OPTIONAL,
  &substrings-match       MATCHING-RULE OPTIONAL,
  &single-valued          BOOLEAN DEFAULT FALSE,
  &collective             BOOLEAN DEFAULT FALSE,
  &no-user-modification   BOOLEAN DEFAULT FALSE,
  &usage                  Attribute-Usage
                            DEFAULT userApplications,
  &id                     OBJECT IDENTIFIER UNIQUE }
WITH SYNTAX {
  [SUBTYPE OF             &derivation]
  [WITH SYNTAX            &Type]
  [EQUALITY MATCHING RULE &equality-match]
  [ORDERING MATCHING RULE &ordering-match]
  [SUBSTRINGS MATCHING RULE &substrings-match]
```

[9]Note the keywords `WITH` and `SYNTAX` are used also *within* the user-friendly definition: these keywords are not forbidden in rule ⟨9⟩ on the preceding page.

```
    [SINGLE VALUE             &single-valued]
    [COLLECTIVE               &collective]
    [NO USER MODIFICATION     &no-user-modification]
    [USAGE                    &usage]
     ID                       &id }


  AttributeUsage ::= ENUMERATED { userApplications(0),
    directoryOperation(1), distributedOperation(2),
    dSAOperation(3) }
```

where `&derivation` indicates that the attribute specializes another one; `&Type` is the ASN.1 type of the attribute's value; `&equality-match`, `&ordering-match` and `&substrings-match` reference the comparison rules to apply on this attribute when consulting the database; `&single-valued` equals `TRUE` if the attribute can have only one value; `&collective` equals `TRUE` if the value of this attribute is shared by several entities of the directory; `&no-user-modification` equals `TRUE` if the value cannot be modified by the directory service users; `&usage` enables to restrict the usage context of this attribute; `&id` is the (single and unique) object identifier of this attribute.

We do not aim at describing the directory service into detail here; are readers interested to know more on this information object class of the X.500 recommendation, they may refer to [Cha96].

The comparison rules are instances of the class `MATCHING-RULE`:

```
  MATCHING-RULE ::= CLASS {
    &AssertionType OPTIONAL,
    &id            OBJECT IDENTIFIER UNIQUE }
  WITH SYNTAX {
    [SYNTAX        &AssertionType]
     ID            &id }
```

They include the type of the data to be compared and a distinct object identifier for every matching rule.

The example of the `ATTRIBUTE` class clearly shows that an information object class is a motley mixture of information which potentially allows defining any concept. We shall see that only part of this information is actually selected depending on the specific communication needs of the case at hand.

Our next example is the attribute which represents an individual name; the comparison function returns true if the two arguments are equal regardless of their (upper/lower) case:

```
name ATTRIBUTE ::= {
  WITH SYNTAX DirectoryString
  EQUALITY MATCHING RULE caseIgnoreMatch
  ID {joint-iso-itu-t ds(5) attributeType(4) 2} }
DirectoryString ::= CHOICE {
  teletexString   TeletexString (SIZE (1..maxSize)),
  printableString PrintableString (SIZE (1..maxSize)),
  universalString UniversalString (SIZE (1..maxSize)),
  bmpString       BMPString (SIZE (1..maxSize)),
  utf8String      UTF8String (SIZE (1..maxSize)) }
maxSize INTEGER ::= 25

caseIgnoreMatch MATCHING-RULE ::= {
  SYNTAX DirectoryString
  ID {id-mr 2} }
id-mr OBJECT IDENTIFIER ::=
  { joint-iso-itu-t ds(5) matchingRule(13) }
```

The [X.501] recommendation defines many other information objects and the reader can find them in the annexes of this recommendation or in [Cha96]. We shall use the examples introduced here in the rest of this chapter.

15.5 Value sets and information object sets

15.5.1 User's Guide

In this section, we describe how to collect objects (values respectively) to create object sets (value sets respectively). These will be used in particular for defining specific subtype constraints, called table constraints.

The simplest way of defining an object set is to enumerate the objects in curly brackets, separated by a vertical bar. An object set is defined by a name that begins with an upper-case letter; it is followed by the name of the object class:

```
MatchingRules MATCHING-RULE ::= {
  caseIgnoreMatch | booleanMatch | integerMatch }
```

In order to make out the structure of an object set, the reader may represent it as a matrix whose columns are the objects of the set and the

	`caseIgnoreMatch`	`booleanMatch`	`integerMatch`
`&AssertionType`	`DirectoryString`	`BOOLEAN`	`INTEGER`
`&id`	`{id-mr 2}`	`{id-mr 13}`	`{id-mr 14}`

Figure 15.2: Object set representation matrix

rows the fields of the class[10]. The collection of objects `MatchingRules` defined above can therefore be represented by the matrix of Figure 15.2.

An object set can also be defined with the set operators `UNION`, `INTERSECTION`, `ALL` and `EXCEPT` encountered in Section 13.11 on page 285 when discussing the subtype constraint combination. Hence, the set:

```
LessMatchingRules MATCHING-RULE ::= {
  MatchingRules EXCEPT caseIgnoreMatch }
```

contains only the two objects `booleanMatch` and `integerMatch`.

If two objects have exactly the same definition but have different names, say `a` and `b`, then they both appear in the object set (i.e. the set `{{a} UNION {b}}` is equivalent to `{a|b}`).

If an object set contains an extension marker "...", the set is dynamically extensible, which implies that the communicating application may add or remove objects while running (see rule ⟨5⟩ on page 332). It is important to note that object sets are the only dynamically extensible entities in ASN.1: types, subtype constraints and value sets can only be extended statically in a new version of the ASN.1 specification).

So the set:

```
ExtensibleMatchingRules MATCHING-RULE ::= {
  caseIgnoreMatch | booleanMatch | integerMatch, ... }
```

initially contains three objects, but some other objects may be dynamically added (and some removed) by the communicating application. Note the syntax of this example: a comma appears before the extension marker while the objects are separated by a vertical bar.

If an object is defined by:

```
ExtensibleMatchingRules MATCHING-RULE ::= {...}
```

its content depends entirely on the implementation: it statically contains

[10]We do not adopt here the convention of the ASN.1 standard where the objects are the matrix *rows*, simply for clarity's sake since our representation will enable us to explain table constraints more visually in the next section. The use of this convention is consistent throughout the rest of this text.

no objects at the beginning but the communicating application will dynamically insert them according to its own needs and a mechanism is supposed to exist independently of ASN.1 so that the peer application can add or remove the same object in its own dynamically extensible object set.

This concept of extensibility would allow, for example, an open network management system (see Section 23.3 on page 482) to dynamically take into account the description of new network elements as information objects. It is just as if the object set were defined within the specification and parameterized this specification. The extensible object set is one case of parameter of the abstract syntax, which will be dealt with in Section 17.3 on page 389.

The definition of a value set is syntactically the same as the object set's: the reference name begins with an upper-case letter but is followed by a type or a type reference:

```
Values INTEGER ::= { 1 | 2 | 3 }
```

This value set is semantically equivalent to the following constrained type:

```
Values ::= INTEGER (1|2|3)
```

This semantical equivalent demands that a value set cannot be empty since a type should contain at least one value.

The notion of value set was introduced in the ASN.1:1994 standard to homogenize the syntax of collections of objects with that of collections of values, and their use in a parameterization context (see Chapter 17) in particular. For other cases, it is recommended to use the usual subtyping (see Chapter 13), which is both more readable and intuitive.

A value set can contain an extension marker, but contrary to object sets, it is never dynamically extensible and has to be statically extended in a new version of the ASN.1 specification. The '{...}' expression cannot, therefore, be used to define a value set, for this set would be initially empty.

15.5.2 Reference Manual

Object set notation

ObjectSet → "{" *ObjectSetSpec* "}"

$$
\begin{array}{lll}
\textit{ObjectSetSpec} & \rightarrow & \textit{RootElementSetSpec} \\
 & | & \textit{RootElementSetSpec} \text{ ``,'' ``...''} \\
 & | & \textit{RootElementSetSpec} \text{ ``,'' ``...'' ``,''} \\
 & & \textit{AdditionalElementSetSpec} \\
 & | & \text{``...''} \\
 & | & \text{``...'' ``,''} \textit{ AdditionalElementSetSpec}
\end{array}
$$

⟨1⟩ An object set is defined using the set operators of *ElementSetSpec* (see on page 335). The objects are generally separated by a vertical bar "|".

⟨2⟩ The set *ObjectSetSpec* is made of the objects referenced by *RootElementSetSpec* and those referenced by *AdditionalElementSetSpec*. The set operators take into account all the information objects, whether they are within the root or in the extensions of each object set used in the set combination (this does not apply to value sets as indicated in rule ⟨14⟩ on the next page).

⟨3⟩ If an extensible object set, `A`, is referenced within the extension root of another extensible object set, `B`, all objects of `A` are objects of the extension root of `B`, whether they are in the root or in the extension series of `A`.

⟨4⟩ If two syntactically identical objects are referenced by two different names in an *ObjectSet*, they must appear twice.

⟨5⟩ If the second or third alternative of *ObjectSetSpec* is used, the set includes objects in the extension root; these objects might be supported by any implementation of the protocol and therefore cannot be dynamically removed by the communicating application[11]. Such a specific behavior, not documented in the ASN.1 standard, needs to be clearly described as a comment in the ASN.1 specification.

⟨6⟩ If the fourth alternative of *ObjectSetSpec* is used, i.e. *ObjectSet* equals "{...}", the object set depends on the implementation: it is initially empty but elements may be dynamically added or removed by the communicating application [ISO8824-2, annex E].

⟨7⟩ If an extensible object set is referenced within another object set, the extension marker is inherited (see also rule ⟨2⟩ on the current page).

⟨8⟩ In the case of a dynamically extensible object set whose objects have a field marked `UNIQUE`, the index of an object that has been removed can be re-used later for another object. Checking this property

[11]For example, this principle is adopted by the X.400 and X.500 standards as well as in secured applications to impose the availability of encryption algorithms on every implementation.

is down to the application. This problem may arise for an index of type `INTEGER`, for instance, but it cannot occur if the index is typed `OBJECT IDENTIFIER`.
⟨9⟩ Contrary to extensions in `CHOICE`, `SEQUENCE` and `SET` types, the version double square brackets "`[[`" and "`]]`", together with the second extension marker "`...`" and the exception marker "`!`" are not allowed in an extensible object set definition.
⟨10⟩ If the module contains the `EXTENSIBILITY IMPLIED` clause in its header, it does not influence the object sets defined in this module (see rule ⟨9⟩ on page 114).

Value set notation

ValueSet → "{" *ElementSetSpecs* "}"
ElementSetSpecs → *RootElementSetSpec*
| *RootElementSetSpec* "," "..."
| *RootElementSetSpec* "," "..." ","
AdditionalElementSetSpec

⟨11⟩ The definitions 'typereference *Type* "`::=`" "{" *ElementSetSpecs* "}"' and 'typereference "`::=`" *Type* "`(`" *ElementSetSpecs* "`)`"' are semantically equivalent.
The first notation (in curly brackets), which is specific to value sets, was introduced to make the definitions of the types parameterized by value sets consistent with the definitions of the types parameterized by object sets. For other cases, it is recommended to use the usual subtyping (i.e. with parentheses), which is both more readable and more intuitive (see Chapter 13).
⟨12⟩ A value set is defined using the set operators of *ElementSetSpec* (see on page 335). Values are generally separated by a vertical bar "`|`".
⟨13⟩ The *ValueSet* is made of the union of the values referenced by *RootElementSetSpec* with those referenced by *AdditionalElementSetSpec*.
⟨14⟩ For *RootElementSetSpec*, and possibly for *AdditionalElementSetSpec*, the set operators are applied to every set's extension root regardless of any extension mark "`...`" or of extensions.
⟨15⟩ The final result of the set combination (*ElementSetSpecs*) must have at least one common value with the parent type or with the parent type's root if the latter includes an extensible subtype constraint (intermediate results of the set combination may be empty). This

condition is necessary because *ValueSet* is a type (see rule ⟨7⟩ on page 110) which ASN.1 imposes to be not empty to be encoded (see the difference with *ObjectSet*, on page 331, which may be empty since it is never encoded).

⟨16⟩ Before applying set operators (according to rule ⟨14⟩ on the page before), every value set (*SubtypeElements*) appearing in the set combination *ElementSetSpecs* must be ensured to have at least one common value with the parent type.

⟨17⟩ Contrary to extensions in `CHOICE`, `SEQUENCE` and `SET` types, the version double square brackets "`[[`" and "`]]`", nor the second extension marker "`...`" nor the exception marker "`!`" are allowed in an extensible value set definition.

⟨18⟩ The `EXTENSIBILITY IMPLIED` clause in the header of a module does not affect the value sets defined in this module (see rule ⟨9⟩ on page 114).

⟨19⟩ Contrary to an object set, a value set is not dynamically extensible (otherwise it could happen to be empty, which is not allowed).

⟨20⟩ If *ElementSetSpecs* references extensible value sets, the extension marker is not inherited by the resulting set (see also rule ⟨14⟩ on the page before). A value set is therefore extensible only if it includes an extension marker 'at its top level'.

⟨21⟩ The ASN.1 working group now thinks it has been a mistake to allow the extension marker "`...`" in value sets because the rule ⟨11⟩ on the preceding page implies that all ASN.1 types can be extensible, which proves troublesome for their PER encoding (see Chapter 20). It is therefore recommended to be very cautious when inserting extension makers in value sets.

Common productions

$$\begin{array}{lll} \textit{RootElementSetSpec} & \rightarrow & \textit{ElementSetSpec} \\ \textit{AdditionalElementSetSpec} & \rightarrow & \textit{ElementSetSpec} \end{array}$$

⟨22⟩ In the rest of this section, the term 'element' stands for either a value or an information object.

⟨23⟩ It is recommended not to use too complex set combinations since an ASN.1 tool is very unlikely to be able to compute them.

⟨24⟩ A (value or object) set of the form {{`a | b`}} is equivalent to {`a | b`}, for there is no concept of set of (value or object) sets in ASN.1 (contrary to mathematical common knowledge).

ElementSetSpec → *Unions*
| `ALL` *Exclusions*

⟨25⟩ The resulting set is made of all the elements specified in *Unions* (1st alternative), or of all the elements of the governor except those specified in *Exclusions* (2nd alternative) (see Figure 13.3 on page 286).

Unions → *Intersections*
| *UElems UnionMark Intersections*

⟨26⟩ The resulting set is made of all the elements specified in *Intersections* (1st alternative), or of those appearing at least once in *UElems* or in *Intersections* (2nd alternative) (see Figure 13.3 on page 286).

UElems → *Unions*
UnionMark → "`|`"
| `UNION`

⟨27⟩ It is recommended to use throughout a specification either the keywords `UNION` and `INTERSECTION`, or the symbols "`|`" and "`^`".

Intersections → *IntersectionElements*
| *IElems IntersectionMark IntersectionElements*

⟨28⟩ The resulting set is made of all the elements specified in *IntersectionElements* (1st alternative) or of those appearing at least once in *IElems* or in *IntersectionElements* (2nd alternative) (see Figure 13.3 on page 286).

IElems → *Intersections*
IntersectionMark → "`^`"
| `INTERSECTION`
IntersectionElements → *Elements*
| *Elems Exclusions*

⟨29⟩ The resulting set is made of all the elements specified in *Elements* (1st alternative), or of those appearing in *Elems* but not in *Exclusions* (2nd alternative) (see Figure 13.3 on page 286).

Elems → *Elements*
Exclusions → `EXCEPT` *Elements*

⟨30⟩ `EXCEPT` takes precedence over `INTERSECTION` and "`^`", which, themselves, take precedence over `UNION` and "`|`".
⟨31⟩ If *ElementSetSpec* is used for defining a value set, the joint use of the `EXCEPT` operator together with a recursive value set cannot define an

empty value set. For example, the type '`U ::= SET { a U (ALL EXCEPT U) OPTIONAL }`', which contains the value {} is not empty. But the constraint '`(ALL EXCEPT U)`' is equivalent to an empty value set, which is not allowed semantically, then the type `U` is illegal.

Elements → *SubtypeElements*
  | *ObjectSetElements*
  | "(" *ElementSetSpec* ")"

⟨32⟩ The alternative *SubtypeElements* (see Section 13.11.2 on page 288) can be used only if *ElementSetSpec* is used to define a value set (see also rule ⟨16⟩ on page 334).
⟨33⟩ The alternative *ObjectSetElements* can be used only if *ElementSetSpec* is used to define an information object set.

ObjectSetElements → *Object*
  | *DefinedObjectSet*
  | *ObjectSetFromObjects*
  | *ParameterizedObjectSet*

⟨34⟩ In the four alternatives, the objects should be instances of the governing class.

15.6 Accessing the information stored in objects and object sets

15.6.1 User's Guide

Since objects or object sets are meant to store information, we need a notation to point at every piece of it so that it could be used when defining types and abstract values in ASN.1 specifications (remember that data can only be encoded, hence transmitted, if they are related to an ASN.1 type). To extract the information from an object (or from an object set), we use a dotted notation which follows the object (or object set) reference.

Thus, the type `DirectoryString` can be extracted of the object `caseIgnoreMatch` (see on page 329) to define, for example, the following value:

```
caseIgnoreMatchValue caseIgnoreMatch.&AssertionType ::=
  printableString:"Escher"
```

We can also extract the object identifier of this very object to write:

```
id-mr-caseIgnoreMatch OBJECT IDENTIFIER ::=
  caseIgnoreMatch.&id
```

And if this dotted notation itself references an object, we go further down in the information by concatenating the fields as in the `value` below:

```
CLASS1 ::= CLASS { &obj CLASS2 }
CLASS2 ::= CLASS { &val INTEGER }
object1 CLASS1 ::= { &obj object2 }
object2 CLASS2 ::= { &val 5 }
value INTEGER ::= object1.&obj.&val
```

Specific conditions apply to this dotted field chain, in particular to avoid intractable self-reference: these are described in the section 'Reference Manual'.

This dotted notation is very common in computing. We now apply it to information object sets to build up collections of objects or values.

To extract the object identifiers of the information object set `MatchingRules` (see on page 329), we simply write:

```
Oids OBJECT IDENTIFIER ::= {MatchingRules.&id}
```

Note the capital letter for `Oids` and the curly brackets after the symbol "`::=`" to indicate that the result of this extraction is a value set (of type `OBJECT IDENTIFIER`). This value set is actually equivalent to:

```
Oids OBJECT IDENTIFIER ::=
  { {id-mr 2} | {id-mr 12} | {id-mr 13} }
```

that is to say that we extract the second row of the matrix in Figure 15.2 on page 330, which could have been *informally* represented as follows to highlight the row that is extracted:

```
Oids OBJECT IDENTIFIER ::=
  { | {id-mr 2} | {id-mr 12} | {id-mr 13} | }
```

In mathematical terms, we would say that we carried out a *projection* of the matrix `MatchingRules` on its `&id` row.

If we now go back on the information object class `OTHER-FUNCTION` (see on page 324) to define the object set:

```
SupportedFunctions OTHER-FUNCTION ::= {
  addition-of-2-integers | substraction-of-2-integers |
  multiplication-of-2-integers }
```

First part[a]	**Last field name**[a]	**Production**[b]
Object	Fixed-type value	*ValueFromObject*
	Variable-type value	*ValueFromObject*
	Fixed-type value set	*ValueSetFromObjects*
	Variable-type value set	*ValueSetFromObjects*
	Type	*TypeFromObject*
	Object	*ObjectFromObject*
	Object set	*ObjectSetFromObjects*
Object set	Fixed-type value	*ValueSetFromObjects*
	Variable-type value	IMPOSSIBLE
	Fixed-type value set	*ValueSetFromObjects*
	Variable-type value set	IMPOSSIBLE
	Type	IMPOSSIBLE
	Object	*ObjectSetFromObjects*
	Object set	*ObjectSetFromObjects*

[a]Let a dotted notation of the form '`obj.&a.&b.&c.&d`', the part '`obj.&a.&b.&c`' is called the *first part* and '`&d`' the *last field name* or 'second part' (it is the object field or object set field pointed at by '`obj.&a.&b.&c`').

[b]These productions of ASN.1 grammar are presented in Section 15.6.2 starting on the next page.

Table 15.2: Information extraction from objects and object sets

then the notation `SupportedFunctions.&Errors` denotes the object set that is in fact the set of all the errors returned by these three functions. It is built up by a union ("`|`" symbol) of the object sets associated with the `&Errors` field of every object in the `SupportedFunctions` set.

If the object set `SupportedFunctions` is dynamically extensible, the resulting object set `SupportedFunctions.&Errors` is dynamically extensible too. If, however, a value set is built by information extraction from an extensible object set, it is not extensible (a value set cannot be dynamically extensible).

All the possible extractions operated on an object or an object set are detailed in Table 15.2. The first row can be read "if we extract a fixed-type value field from an object, we obtain a value provided we conform to the rules associated with the grammar production *ValueFromObject* described in the Reference Manual further on".

15.6.2 Reference Manual

⟨1⟩ The series of dotted fields '*ReferencedObjects* "." *FieldName*' (where the production *ReferencedObjects* and *FieldName* are defined on page 341) used throughout the current section can be divided into two parts: the *first part* (*i.e.* without the last *PrimitiveFieldName*) and the *last field name* or 'second part'. For example, for the reference '`obj.&a.&b.&c.&d`', the first part is '`obj.&a.&b.&c`' and the last field name is '`&d`'.
⟨2⟩ The Table 15.2 on the preceding page indicates which grammar production must be applied depending on what is referenced by the first part and the last field name of a chain of field names.

ValueFromObject notation

ValueFromObject → *ReferencedObjects* "." *FieldName*

⟨3⟩ The first part should denote an object and the last field name should denote a fixed-type or variable-type value field of this object, which means that this notation is used to extract a value from an information object.
⟨4⟩ If *FieldName* is marked `OPTIONAL` in the class definition, this notation can only be associated with a field of an object that is marked `OPTIONAL` or `DEFAULT` in its information object class, or to a component that is marked `OPTIONAL` or `DEFAULT` in a `SEQUENCE` or `SET` type.

ValueSetFromObjects notation

ValueSetFromObjects → *ReferencedObjects* "." *FieldName*

⟨5⟩ The first part denotes:
- an object; then the last field name denotes a fixed-type or variable-type value set field of this object, or
- an object set; then the last field name denotes a fixed-type value field or a fixed-type value set field of any object in this set;

that is to say, this notation can extract a value set from an information object set (a matrix) or from an object (a column of a matrix).
⟨6⟩ If the first part denotes an object and if the last field name denotes a value set field, *ValueSetFromObjects* is the union of the selected value sets (every value can appear only once in the resulting set).
⟨7⟩ If the first part denotes a object set and if the last field name denotes a fixed-type value field, *ValueSetFromObjects* is the set made of the selected values.

⟨8⟩ If a value set *ValueSetFromObjects* is extracted from an extensible object set, the resulting value set does not inherit the extension marker (see rule ⟨21⟩ on page 334). As a result, it is impossible to use a dynamically extensible object set of the form '{...}' or '{..., *AdditionalElementSetSpec*}' (see production *ObjectSetSpec* page 332) because the extracted value set can be empty sometime during communication.
⟨9⟩ If *FieldName* is marked `OPTIONAL` in the class definition, this notation can only be associated with a field of an object that is marked `OPTIONAL` or `DEFAULT` in its information object class, or to a component that is marked `OPTIONAL` or `DEFAULT` in a `SEQUENCE` or `SET` type.

***TypeFromObject* notation**

TypeFromObject → *ReferencedObjects* "." *FieldName*

⟨10⟩ The first part must denote an object and the last field name must denote a type field of this object, which means that this notation is used to extract a type from an information object.
⟨11⟩ If *FieldName* is marked `OPTIONAL` in the class definition, this notation can only be associated with a field of an object that is marked `OPTIONAL` or `DEFAULT` in its information object class, or to a component that is marked `OPTIONAL` or `DEFAULT` in a `SEQUENCE` or `SET` type.
⟨12⟩ This production, which extracts a precise type from an information object, should not be confused with the production *ObjectClassField-Type* on page 347 that defines an open type.

***ObjectFromObject* notation**

ObjectFromObject → *ReferencedObjects* "." *FieldName*

⟨13⟩ The first part must denote an object and the last field name must denote an object field of this object, i.e. this notation is used to extract an object from another information object.
⟨14⟩ If *FieldName* is marked `OPTIONAL` in the class where it is defined, this notation can only be associated with a field of an object that is marked `OPTIONAL` or `DEFAULT` in its information object class.

***ObjectSetFromObjects* notation**

ObjectSetFromObjects → *ReferencedObjects* "." *FieldName*

⟨15⟩ The first part denotes:

- an object; then the last field name indicates an object set field of this object, or
- a object set; then the last field name indicates an object field or an object set field of any object in this set;

that is to say this notation can extract an object set from another information object set (a matrix) or from an information object (a column of a matrix).

⟨16⟩ If the first part denotes an object set and if the last field name denotes an object set field, *ObjectSetFromObjects* is the union of the selected object sets (see rule ⟨4⟩ on page 332).

⟨17⟩ If *FieldName* is marked `OPTIONAL` in the class definition, this definition can only be associated with a field of an object that is marked `OPTIONAL` or `DEFAULT` in its information object class.

⟨18⟩ If an extensible object set is referenced within another object set, the extension marker is inherited, i.e. the notation *ObjectSet-FromObjects* produces a (dynamically) extensible object set if either the pointed object set or one of the extracted object sets is extensible (note the difference with rule ⟨8⟩ on the preceding page).

Productions communes

$$\begin{aligned} \textit{ReferencedObjects} &\rightarrow \underline{\textit{DefinedObject}} \quad | \ \underline{\textit{ParameterizedObject}} \\ &\quad | \ \underline{\textit{DefinedObjectSet}} \ | \ \underline{\textit{ParameterizedObjectSet}} \\ \textit{FieldName} &\rightarrow \textit{PrimitiveFieldName} \text{ “.” } \cdots^{+} \end{aligned}$$

⟨19⟩ In *FieldName*, all the *PrimitiveFieldName*s but the last one must be objectfieldreferences or objectsetfieldreferences.

PrimitiveFieldName → typefieldreference
| valuefieldreference | valuesetfieldreference
| objectfieldreference | objectsetfieldreference

15.7 A simple case study of how to extract information modeled by a class

15.7.1 User's Guide

After this long description of the concepts of class, object and object set, we now address the real issue for those involved in telecommunications: during a data transfer, how can we use the information stored in objects? This is where the quotation due to Rousseau in the epigraph of this chapter starts to take its full meaning!

Consider a directory where every entry describes an individual by its surname, first name and phone number. Then these three attributes can be modeled by three information objects[12] of the class `ATTRIBUTE` defined on page 327:

```
surname ATTRIBUTE ::= {      -- family name
  SUBTYPE OF  name
  WITH SYNTAX DirectoryString
  ID          id-at-surname }
givenName ATTRIBUTE ::= {    -- first name
  SUBTYPE OF  name
  WITH SYNTAX DirectoryString
  ID          id-at-givenName }
countryName ATTRIBUTE ::= {  -- country
  SUBTYPE OF   name
  WITH SYNTAX  PrintableString (SIZE (2)) -- [ISO3166] codes
  SINGLE VALUE TRUE
  ID           id-at-countryName}
```

and be gathered together in the following object set (note the upper-case letter for the reference):

```
SupportedAttributes ATTRIBUTE ::=
  {surname | givenName | countryName}
```

To modify one of the three attributes' values in our directory, we may assume that it is necessary to transmit the (unique) identifier of this attribute (the `&id` field of the object that models this attribute) and its new value (which should conform to the type stored in the `&Type` field of the *same*[13] object). We collect these data to be transmitted in a `SEQUENCE` type as follows:

```
AttributeIdAndValue1 ::= SEQUENCE {
  ident ATTRIBUTE.&id,
  value ATTRIBUTE.&Type }
```

which can be read: in a value of type `AttributeIdAndValue1`, the `ident` component takes the object identifier stored in the `&id` field[14] of an object

[12]These three information objects are defined in the [X.520] recommendation. The type `DirectoryString` was defined on page 329.

[13]We will see further why we stress on the word 'same'.

[14]The syntax might seem sibylline since the `&id` field does not begin with an upper-case letter whereas a type reference is expected here. The meaning of each of these

of class `ATTRIBUTE` and the `value` component has the type specified by the `&Type` field of an object of class `ATTRIBUTE`.

The `value` component of the `AttributeIdAndValue1` type can therefore have any type depending on the object considered (`DirectoryString` for the `surname` and `givenName` objects, or `PrintableString` for the `countryName` object). The `value` component is said to be of an *open type*. We now have the equivalent for the `ANY` type of ASN.1:1990 (see Section 12.8 on page 241), which has been removed from the ASN.1 standard[15] since 1994.

As for the `ANY` type, a value of an open type is defined by the effective type of the value followed by the "`:`" symbol and the value expression, such as '`INTEGER:5`'.

We propose to use the following *informal* representation for the `AttributeIdAndValue1` type to indicate more clearly which value set corresponds to every component of the `SEQUENCE` type:

```
AttributeIdAndValue1 ::= SEQUENCE {
  ident  all &id fields of the infinity of objects of class ATTRIBUTE ,
  value  the infinity of types potentially definable in ASN.1 }
```

But these two sets are infinite, which is (much!) too large for the initial problem. Besides, a value of an open type is encoded as an octet string (see Part III on page 391) and if the open type is not constrained, the receiving application will not be able to interpret this series of octets.

We now constrain every component of the `SEQUENCE` type so that the information objects that are being considered can be taken in the set `SupportedAttributes` but not in the infinity of objects of the `ATTRIBUTE` class that could be defined. This particular constraint called *simple table constraint*[16] consists in a constraining set denoted in round brackets after the type of every component like all the ASN.1 subtype constraints:

```
AttributeIdAndValue2 ::= SEQUENCE {
  ident ATTRIBUTE.&id({SupportedAttributes}),
  value ATTRIBUTE.&Type({SupportedAttributes}) }
```

notations are given by the semantic rules of the *ObjectClassFieldType* grammar production in Section 15.7.2 on page 347.

[15]In ASN.1:1990, the `AttributeIdAndValue1` type above would have been written:

```
AttributeIdAndValue1 ::= SEQUENCE {
  ident OBJECT IDENTIFIER,
  value ANY }
```

[16]It is defined in [ISO8824-2, clause 10] by the grammar production *SimpleTableConstraint* (see on page 349).

Note, however, the compulsory curly brackets surrounding `SupportedAttributes`, which remind us that the constraint is an object set rather than a constraint by type inclusion (whose syntax imposes a word beginning with an upper-case letter in round brackets, see Section 13.3 on page 261).

The `AttributeIdAndValue2` type is now *informally* represented as follows[17]:

```
AttributeIdAndValue2 ::= SEQUENCE {
  ident | id-at-surname   | id-at-givenName | id-at-countryName |,
  value | DirectoryString | DirectoryString | PrintableString   |}
```

In mathematical terms, we would say that we operated a *projection* of the `SupportedAttributes` matrix of Figure 15.2 on page 330 on its row `&id` and, in parallel, on its row `&Type`. Using the collection and extraction notations of Section 15.5 on page 329, we could similarly define the `AttributeIdAndValue2` type as follows:

```
AttributeIdAndValue2 ::= SEQUENCE {
  ident SupportedAttributes.&id,
  value SupportedAttributes.&Type }
```

Even though the latter notation seems slightly more natural, the `AttributeIdAndValue2` type does not fix the initial problem for all that. Indeed, it is still allowed to choose a different information object (a different column of the matrix) for the information associated with the `ident` component as for the information associated with the `value` component. In other words, transmitting the following value is perfectly valid:

```
value AttributeIdAndValue2 ::= {
  ident id-at-countryName,
  value DirectoryString:universalString:"$$Escher$$" }
```

But the `countryName` attribute selected by the `ident` component only accepts `PrintableString`s of at most two characters whereas the transmitted value is typed `UniversalString` (the value `"$$Escher$$"` contains more than two characters and the "`$`" character does not belong to the `PrintableString` type).

[17]It is only for simplicity's sake that we do not represent the subtype constraints on the `DirectoryString` and `PrintableString` types.

We now have to indicate, using the dedicated ASN.1 notation called *component relation contraint*[18], that when an information object (one of the matrix column) is chosen for the `ident` component of type `AttributeIdAndValue2`, then the same object (column) should be used for its `value` component.

This is represented by the at-sign[19] "`@`", followed by the name of the first component of the `SEQUENCE` type (i.e. `ident`); the symbol and the component name are placed in curly brackets after the object set {`SupportedAttributes`}, which constraints both components:

```
AttributeIdAndValue3 ::= SEQUENCE {
  ident ATTRIBUTE.&id({SupportedAttributes}),
  value ATTRIBUTE.&Type({SupportedAttributes}{@ident}) }
```

In mathematical terms, we would say that this component relation constraint is equivalent to projecting the matrix on one[20] of its columns (remember each column contains the fields of the same object). The result of the horizontal projection of the `SupportedAttributes` matrix on the `&id` row (`&Type` respectively), followed by the vertical projection on one of its columns could be *informally* represented by:

```
val AttributeIdAndValue3 ::= {
  ident  id-at-countryName ,
  value  PrintableString:"F"  }
```

It is a *value* that is represented here informally (and not a type as in all the previous examples) because the vertical projection `@ident` occurs at the encoding stage when an information object is chosen. The `AttributeIdAndValue3` type has the advantage of being completely independent from the attribute whose value is to be changed (hence also independent from the type of this value).

[18]This is defined by the gramar production *ComponentRelationConstraint* on page 350.

[19]This notation should not be confused with the meta-notation defined on page 231, which can be used only in comments.

[20]We could, of course, have equally written:

```
AttributeIdAndValue4 ::= SEQUENCE {
  ident ATTRIBUTE.&id({SupportedAttributes}{@value}),
  value ATTRIBUTE.&Type({SupportedAttributes}) }
```

But it seems more sensible and natural to choose an attribute by its single identifier rather than by the value that it is given. Besides, one column of the matrix can only be selected at a time since the `&id` field is marked `UNIQUE`. The type `AttributeIdAndValue3` is actually equivalent to the type `AttributeTypeAndValue` of the [X.501] recommendation; we only did a bit of renaming to improve readability.

An ASN.1 compiler can, of course, employ the data stored in information object sets to generate a decoder that would check the validity of the messages received (which means that, in our case, the decoder can make sure the value of the `value` component conforms to the type referenced by the `ident` component).

If the object set `SupportedAttributes` includes an extension marker[21] "...", then it is dynamically extensible, i.e. the number of columns together with the horizontal projections `ATTRIBUTE.&id({SupportedAttributes})` and `ATTRIBUTE.&Type({SupportedAttributes})` dynamically change (but the columns that appear before the extension marker cannot be removed dynamically, for they correspond to information objects that should be supported whatever the implementation of the protocol involved).

The component relation constraint `@ident` is actually the equivalent of the former mechanism `ANY DEFINED BY` of ASN.1:1990, removed from the standard in 1994. In ASN.1:1990, the `AttributeIdAndValue3` type below was written:

```
AttributeIdAndValue3 ::= SEQUENCE {
  ident OBJECT IDENTIFIER,
  value ANY DEFINED BY ident }

--        ident       |      value
-- ==================|============================
-- id-at-surname      | DirectoryString
-- id-at-givenName    | DirectoryString
-- id-at-countryName  | PrintableString (SIZE (2))
```

But as this notation did not allow the formal specification of the object set[22] involved (the matrix), the ASN.1:1990 standard recommended to indicate this information in comments with the inconveniences that would ensue: a comment is neither formal nor meant to define a standard; it is not taken into account by compilers[23] (even macro instances were not always used by compilers and were highly un-recommended);

[21]Such is the case for the [X.501] recommendation, in fact, which defines the object set `SupportedAttributes` as: `SupportedAttributes ATTRIBUTE ::= {objectClass|aliasedEntryName, ...}`.

[22]At the time, we should have said the 'set of macro instances' (see Chapter 16).

[23]The implementation of the matrix used to be specific to each tool, which limited the specification portability.

this mechanism is generally static, which does not authorize dynamically extensible matrices or different matrices in every implementation.

We now focus on the detail of the ASN.1 grammar constructions that make it possible to extract the information from information object classes in order to use it in the specification of the data to be transmitted. This presentation continues in Section 15.8 on page 352, where more complex examples are tackled.

15.7.2 Reference Manual

Type notation

ObjectClassFieldType → *DefinedObjectClass* "." *FieldName*

FieldName → *PrimitiveFieldName* "." $\cdots^{+}$

PrimitiveFieldName → typefieldreference
| valuefieldreference | valuesetfieldreference
| objectfieldreference | objectsetfieldreference

⟨1⟩ The first *PrimitiveFieldName* must be a field of the class referenced by *DefinedObjectClass*.
⟨2⟩ The class name *DefinedObjectClass* cannot be followed by an actual parameter list (in curly brackets) before the dot.
⟨3⟩ If *ObjectClassFieldType* denotes a type field, a variable-type value field or a variable-type value set field, this notation defines an *open type* (its value set is the set of all the values that can be specified in ASN.1; it is therefore the equivalent of ASN.1:1990 `ANY` type, which has been removed from the standard since ASN.1:1994). It is recommended to constraint an open type with a constraint using an object set (see rule ⟨20⟩ on page 350) ; if the open type is not constrained, a decoder can only deliver an octet string to the communicating application. This notation cannot be used directly or indirectly when defining the type of a value field or of a value set field of another class; it means the type of a value field or value set field of a class cannot depend on a type field of another class.
⟨4⟩ If *ObjectClassFieldType* denotes a type field, a variable-type value field or a variable-type value set field, this notation has no determined tag and cannot be used in a construction where the tags are required to be distinct (except if it is preceded by a tag explictly inserted by the specifier or if the module has the `AUTOMATIC TAGS` clause in its header).

Moreover, this notation cannot be tagged in `IMPLICIT` mode because the effective type can be any ASN.1 type (see rules ⟨10⟩ on page 223, ⟨10⟩ on page 228 and ⟨7⟩ on page 238).
⟨5⟩ If *ObjectClassFieldType* denotes a fixed-type value field or a fixed-type value set field, this notation gives the type of the field appearing in the class definition.
⟨6⟩ *ObjectClassFieldType* cannot denote an object field or an object set field.
⟨7⟩ If there exists a dotted chain *ObjectClassFieldType* such that the first field is in the class being defined, that all the following fields belong to the class referenced by the previous field and that the last field belongs to the class being defined, then one of the fields at least must be marked `OPTIONAL` or `DEFAULT`. This rule prevents recursive definition of a class that would have no object with a finite representation.
⟨8⟩ *ObjectClassFieldType* can be constrained by a single value (production *SingleValue* on page 261) and by type inclusion (production *ContainedSubtype* on page 263). If *ObjectClassFieldType* is an open type (see rule ⟨3⟩ on the preceding page), it can be constrained by a type (production *TypeConstraint* on page 352).

Value notation

$$\begin{array}{rcl} \textit{ObjectClassFieldValue} & \rightarrow & \textit{OpenTypeFieldVal} \\ & | & \textit{FixedTypeFieldVal} \end{array}$$

⟨9⟩ If the corresponding *ObjectClassFieldType* is a type field, a variable-type value field or a variable-type value set field (see rule ⟨2⟩ on page 317), the alternative *OpenTypeFieldVal* must be used.
⟨10⟩ If the corresponding *ObjectClassFieldType* is a fixed-type value field or a fixed-type value set field, the alternative *FixedTypeFieldVal* must be used.

$$\textit{OpenTypeFieldVal} \rightarrow \underline{\textit{Type}} \text{ ":" } \underline{\textit{Value}}$$

⟨11⟩ *OpenTypeFieldVal* is equivalent to a value of type `ANY` in ASN.1:1990 (see Section 12.8 on page 241 and rule ⟨3⟩ on the preceding page).
⟨12⟩ *Value* must be of type *Type* or of a type that is compatible with *Type* according to the semantic model of ASN.1 (see Section 9.4 on page 121).

$$\begin{array}{rcl} \textit{FixedTypeFieldVal} & \rightarrow & \underline{\textit{BuiltinValue}} \\ & | & \underline{\textit{ReferencedValue}} \end{array}$$

⟨13⟩ The productions *BuiltinValue* and *ReferencedValue* are defined on page 109.
⟨14⟩ The value (*BuiltinValue* or *ReferencedValue*) must be of the type specified by the corresponding field in the class definition or of a type that is compatible with it according to the semantic model of ASN.1 (see Section 9.4 on page 121).

Subtype constraints

Constraint → "`(`" *ConstraintSpec* *ExceptionSpec* "`)`"

⟨15⟩ The production *ExceptionSpec* is defined on page 255. It can be used only if the object set referenced in a *SimpleTableConstraint* and in a *ComponentRelationConstraint* is a parameter of the current assignment (see rule ⟨3⟩ on page 293).

ConstraintSpec → *ElementSetSpecs*
| *GeneralConstraint*

GeneralConstraint → *UserDefinedConstraint*
| *TableConstraint*
| *ContentsConstraint*

TableConstraint → *SimpleTableConstraint*
| *ComponentRelationConstraint*

⟨16⟩ A *TableConstraint* can only be applied to an *ObjectClassFieldType* (see next rules) and to the `INSTANCE OF` type (see ⟨5⟩ on page 359).

SimpleTableConstraint → *ObjectSet*

⟨17⟩ *ObjectSet* is an object set of the class referenced at the beginning of the constrained *ObjectClassFieldType* (see also rule ⟨31⟩ on page 351). It necessarily appears in curly brackets (even if it is a reference to an object set *DefinedObjectSet*).
⟨18⟩ The last *FieldName* of the constrained *ObjectClassFieldType* is used for selecting a row in the associated matrix. Indeed, an abstract table can be associated with an information object or an information object set; the rows of this table are the fields of the object, and the columns are the objects considered.
⟨19⟩ If one of the fields of the information object class references this very class, the associated table can have an infinity of rows. This is allowed a priori.

⟨20⟩ For a type field (see rule ⟨2⟩ on page 317), the component value is constrained to conform to one of the types in this row. For a value field, the component value is constrained to take one of the values in this row. For a value set field, the component is constrained to take one of the values in one of the value sets in this row.
⟨21⟩ The sets selected in the previous rule must be ensured to have at least one value so that the rules ⟨32⟩, ⟨33⟩ and ⟨34⟩ on the next page are respected.
⟨22⟩ If *ObjectSet* is an extensible object set, the constrained type does not inherit the extension marker.

ComponentRelationConstraint →
  "{" *DefinedObjectSet* "}" "{" *AtNotation* "," $\cdots^{+}$ "}"

⟨23⟩ A *ComponentRelationConstraint* can be applied only to an *ObjectClassFieldType* (see on page 347) extracted from a class and included (not necessarily at the top level) in another constructed type such as `SEQUENCE` or `SET`, that textually contains all the components identified by the *AtNotation* (see also rule ⟨31⟩ on the next page).

AtNotation → "`@`" *ComponentIdList*
  | "`@.`" *ComponentIdList*

⟨24⟩ The *AtNotation* can appear only in a `SEQUENCE` or `SET` structure. It enables a component of a structure to depend on the value of another component of this structure or of a higher level structure.
⟨25⟩ The *AtNotation* should not be confused with the meta-notation *AbsoluteReference* of the form "`@`modulereference`.`typereference`.`identifier" used in a description text outside an ASN.1 module or in comments for referencing a component of a constructed type (see on page 231).
⟨26⟩ In the first alternative (the "`@`" symbol is not followed by a dot), the parent structure in which the first identifier must be found is the `SEQUENCE`, `SET` or `CHOICE` type that textually encloses the *AtNotation* at the outermost level (examples are given in [ISO8824-3, clause 10.10] and on page 354).
⟨27⟩ In the second alternative (the "`@`" symbol is followed by a dot), the parent structure in which the first identifier must be found is the `SEQUENCE` or `SET` type that textually encloses the *AtNotation* at the innermost level (refer to the example for the `Authentication-value` type on page 356).

ComponentIdList $\rightarrow$ identifier "." $\cdots^{+}$

⟨28⟩ If an identifier denotes a component (an alternative respectively) of a `SEQUENCE` or `SET` type (`CHOICE` type respectively), the next identifier in the list must be one of those appearing in the component list (alternative list respectively) of this type.

⟨29⟩ If an identifier denotes a component that is not of type `SEQUENCE`, `SET` or `CHOICE`, it must be the last element of the *ComponentIdList*.

⟨30⟩ From now on, we call *referencing component*, a component followed by an *AtNotation* in a `SEQUENCE` or `SET` type, and *referenced component*, the component pointed at by this *AtNotation*.

⟨31⟩ The referencing component and all the referenced components must be *ObjectClassFieldType*s (see on page 347) extracted from the same information object class. The *DefinedObjectSet* (in curly brackets) must be the same in the *ComponentRelationConstraint*s as in the *SimpleTableConstraint*s and must be of the same information object class as the one from which the *ObjectClassFieldType*s are extracted.

⟨32⟩ If the referencing component is marked `OPTIONAL` or `DEFAULT` in the structured type definition and is absent in the value, the *ComponentRelationConstraint* is always satisfied (see rule ⟨21⟩ on the preceding page).

⟨33⟩ If the referenced component is marked `OPTIONAL` or `DEFAULT` in the structured type definition and is absent in the value, this value does not satisfy the *ComponentRelationConstraint* unless the referencing component is also marked `OPTIONAL` or `DEFAULT` in the type definition and is absent in the value too.

⟨34⟩ If all the referenced components are present and if the referencing component is present, the constraint is satisfied only if there exists one or more objects in the object set such that, for all fields, every referenced component that is followed by a value field takes the value of the corresponding field in the selected object and every referenced component that is followed by a value set field takes one of the values of the corresponding field in the selected object.

⟨35⟩ If an *ObjectClassFieldType* is constrained by one or more *TableConstraint*s and if *FieldName* denotes a type field, a variable-type value field or a variable-type value set field, there can only be one selected object in the object set if one of the referenced components is an identifier field marked `UNIQUE` (see rule ⟨7⟩ on page 318).

TypeConstraint → *Type*

⟨36⟩ This subtype constraint applies only on an open type. The only open type of ASN.1 is *ObjectClassFieldType* when it denotes a type field, a variable-type value field or a variable-type value set field (see rule ⟨3⟩ on page 347). In that case, the set of values of the constrained open type is the intersection of the set of values of the open type instance and the set of values of the *Type* in the constraint.
⟨37⟩ The production *TypeConstraint* is syntactically equivalent to the production *ContainedSubtype* (see on page 263) where the ϵ alternative of the *Includes* production is retained, but semantically a *TypeConstraint* can only be applied on an open type.
⟨38⟩ Although the permitted values of the open type are semantically those of *Type*, these values are encoded according to the rules associated with the open type, which implies that they will all be preceded, either in BER or PER encoding, by a length field (see Sections 18.2.22 on page 412 and 20.6.11 on page 445).

15.8 More complex examples of information extraction

The notation of component relation constraint (denoted by the "`@`" symbol) is in fact more powerful that the `ANY DEFINED BY` construction of ASN.1:1990 already discussed on page 346, because it can link a `SEQUENCE` or `SET` type component to another component defined in a higher-level structured type but also because several relation constraints can be applied to the same component if one constraint (that is one vertical projection of the matrix) is not enough to reference the information unambiguously. We shall illustrate this point with a few examples.

When several `SEQUENCE`, `SET`, `CHOICE`, `SEQUENCE OF` and `SET OF` types are nested in one another, the "`@`" notation should indicate unambiguously the component of a constructed type. To do so, in a dotted chain of the form "`@ident1.ident2.ident3`", `ident1` is considered as an identifier that belongs to the `SEQUENCE`, `SET` or `CHOICE` type of highest level in the breakdown structure of the considered type assignment; `ident2` is then considered as an identifier appearing in the type associated with `ident1` and finally `ident3` is an identifier appearing in the type associated with `ident2`. If the "`@`" symbol is followed by a dot "." as in "`@.ident`", the `ident` identifier should belong to the first `SEQUENCE` or `SET` type[24] that includes this notation at the lowest level.

The type `AttributeIdAndValue3` on page 345 can, therefore, be equally written:

```
AttributeIdAndValue3 ::= SEQUENCE {
  ident ATTRIBUTE.&id({SupportedAttributes}),
  value ATTRIBUTE.&Type({SupportedAttributes}{@.ident})}
```

since the `SEQUENCE` type includes the definition both at the highest and lowest levels. The same remark applies on the type:

```
AttributeIdsAndValues ::= SET OF SEQUENCE {
  ident ATTRIBUTE.&id({SupportedAttributes}),
  value ATTRIBUTE.&Type({SupportedAttributes}{@.ident})}
```

since the outermost `SET OF` type is not taken into account when the "`@`" symbol is followed by a dot.

Let us now consider the type:

```
AttributeValueAssertion ::= SEQUENCE {
  type      ATTRIBUTE.&Id({SupportedAttributes}),
  assertion ATTRIBUTE.&equality-match.&AssertionType
              ({SupportedAttributes}{@type}) }
```

in which the string[25] `ATTRIBUTE.&equality-match.&AssertionType` will point at the type stored in the `&AssertionType` field of an object of class `MATCHING-RULE`, which is in turn stored by the `&equality-match` field of an object of class `ATTRIBUTE`. The `AttributeValueAssertion` type will define propositions on the attribute's value such as: 'the attribute identified by `type` whose value equals `value` according to the equality matching-test function associated with this attribute and defined by `ATTRIBUTE.&equality-match`'.

[24]Note that a `CHOICE` type is not allowed here. Indeed, it is impossible to write anything like:

```
CHOICE { alt1 ATTRIBUTE.&id({SupportedAttributes}),
         alt2 ATTRIBUTE.&value({SupportedAttributes}{@.alt1})}
```

because if the `alt2` alternative is selected, then the `alt1` alternative is not selected so that its value cannot be used for constraining `alt2`.

[25]This string conforms to the grammar production *ObjectClassFieldType* presented on page 347.

The `AttributeValueAssertion` type is used in the item filter definition for search routines in the X.500 directory:

```
FilterItem ::=  CHOICE {
  equality         [0] AttributeValueAssertion,
  substrings       [1] SEQUENCE {
    type    Attribute.&id({SupportedAttributes}),
    strings SEQUENCE OF CHOICE {
      initial  [0] ATTRIBUTE.&Type
        ({SupportedAttributes}{@substrings.type}),
      any      [1] ATTRIBUTE.&Type
        ({SupportedAttributes}{@substrings.type}),
      final    [2] ATTRIBUTE.&Type
        ({SupportedAttributes}{@substrings.type}) }},
  greaterOrEqual   [2] AttributeValueAssertion,
  lessOrEqual      [3] AttributeValueAssertion,
  present          [4] AttributeType,
  approximateMatch [5] AttributeValueAssertion,
  extensibleMatch  [6] MatchingRuleAssertion }
```

The `@substrings.type` notation indicates the highest-level `type` component in the `substrings` alternative of the top-level `CHOICE`.

These few examples should have convinced the reader of the flexibility of relation constraints defined with the at-sign "`@`". We still have to demonstrate that several component relation constraints can be applied to extract a single object of an object set when one relation constraint is not sufficient to reference the information unambiguously.

For this, let us consider the following type, which allows to change all the attributes for which the `&usage` field equals a given value:

```
Attribute-desc ::= SEQUENCE {
  usage ATTRIBUTE.&usage({SupportedAttributes}),
  list  SEQUENCE OF SEQUENCE {
    ident ATTRIBUTE.&id({SupportedAttributes}{@usage}),
    value ATTRIBUTE.&Type
            ({SupportedAttributes}{@usage,@.ident}) }}
```

We choose the value for the `usage` component, say `userApplications` by default (the `&usage` field is of the enumerated type `AttributeUsage` defined on page 328).

Several objects may store this value in their `&usage` field; it is actually the case for the two objects `objectClass` and `aliasedEntryName` of the object set `SupportedAttributes` (see on page 346) since they should

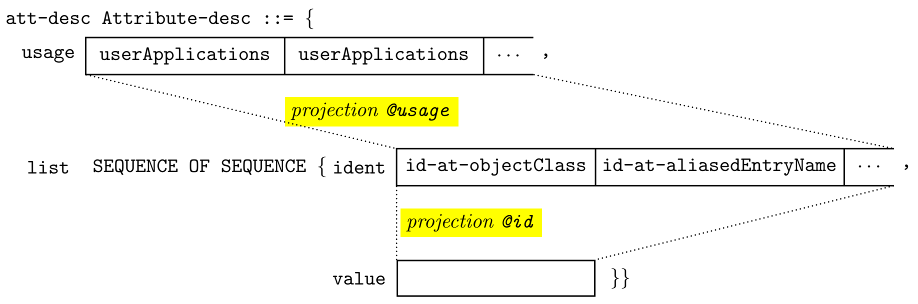


Figure 15.3: Double projection mechanism of two component relation constraints

appear in any implementation of the directory service. The component relation constraint `@usage`, therefore, does not always select one object at a time in the `SupportedAttributes` set (otherwise said, the projection on the `@usage` row of the `SupportedAttributes` matrix may return several items with the same value). In this first selection of objects, there should be considered a second selection by the relation constraint `@ident`, which necessarily gives one object since the `&id` field is an identifier field of class `ATTRIBUTE` (it is followed by the `UNIQUE` marker).

This double selection can be represented by Figure 15.3 (remember that the `SupportedAttributes` object set is dynamically extensible). So, a value of `Attribute-desc` type could be for instance:

```
att-desc Attribute-desc ::= {
  usage userApplications,
  list  { { ident id-at-objectClass,
            value oid },
          { ident id-at-aliasedEntryName,
            value distinguishedName }}}
```

15.9 The pre-defined `TYPE-IDENTIFIER` class and `INSTANCE OF` type

The `TYPE-IDENTIFIER` class sets up an information association frequently used in specifications. It is standardized in [ISO8824-2, annexes A and B] and can, therefore, be directly used within a module as any ASN.1 pre-defined concept.

15.9.1 User's Guide

The `TYPE-IDENTIFIER` class is surely one of the simplest of its kind: it associates a universally-unique object identifier (see Section 10.8 on page 153) with any ASN.1 type. It is defined with a user-friendly syntax as follows:

```
TYPE-IDENTIFIER ::= CLASS {
  &id   OBJECT IDENTIFIER UNIQUE,
  &Type }
WITH SYNTAX {&Type IDENTIFIED BY &id}
```

This information association by means of a pair ⟨object identifier, ASN.1 type⟩ is very often used, particularly to replace the `ANY` type of ASN.1:1990 (see Section 12.8 on page 241). In fact, such a model is necessary each time an ASN.1 specification is 'gapped' by data types to be agreed on when transmitting them. We have used a similar model in the `ATTRIBUTE` class defined on page 327 because the type of the value associated with an attribute depends on this very attribute.

In the Association Control Service Element standard (ACSE [ISO8650-1]), an object of class `TYPE-IDENTIFIER` can associate an ASN.1 type with the object identifier of the authentication mechanism used when establishing the association:

```
MECHANISM-NAME ::= TYPE-IDENTIFIER
```

The abstract syntax of an authentication value is determined by the authentication mechanism using a component relation constraint in which the "`@.`" symbol indicates that the identifier `other-mechanism-name` belongs to the type that includes this constraint at the lowest level, i.e. the `SEQUENCE` type:

```
Authentication-value ::= CHOICE {
  charstring [0] IMPLICIT GraphicString,
  bitstring  [1] BIT STRING,
  external   [2] EXTERNAL,
  other      [3] IMPLICIT SEQUENCE {
    other-mechanism-name  MECHANISM-NAME.&id({ObjectSet}),
    other-mechanism-value MECHANISM-NAME.&Type
      ({ObjectSet}{@.other-mechanism-name}) }}
```

An ASN.1 value referenced in an object of class TYPE-IDENTIFIER is transmitted using an open type and extracting the information as indicated in Section 15.6 on page 336:

```
SEQUENCE { type-id TYPE-IDENTIFIER.&id,
           value   [0] EXPLICIT TYPE-IDENTIFIER.&Type }
```

But as such information extractions frequently occur, ASN.1 provides the adequate pre-defined INSTANCE OF type to achieve them. A type of the form 'INSTANCE OF *DefinedObjectClass*' is therefore defined as follows:

```
SEQUENCE { type-id DefinedObjectClass.&id,
           value   [0] EXPLICIT DefinedObjectClass.&Type }
```

Note the tag in explicit mode[26] before the open type of the value component to make sure that the tags of the components of the SEQUENCE are always distinct (this value component could otherwise have any tag when it is allocated a value).

The INSTANCE OF type only extracts information from objects of class TYPE-IDENTIFIER and it is recommended, as indicated in Section 15.6 on page 336, to state the object set involved. For this, ASN.1 permits that a table constraint could be associated with the INSTANCE OF type (although normally such a constraint can only follow a field extracted from a class). A type of the form 'INSTANCE OF *DefinedObjectClass* ({*ObjectSet*})' is developed in:

```
SEQUENCE { type-id DefinedObjectClass.&id ({ObjectSet}),
           value   [0] DefinedObjectClass.&Type
                         ({ObjectSet}{@.type-id}) }
```

where the component relation constraint @.type-id on the open type of the component value should be noted. We have here the equivalent of the ANY DEFINED BY type withdrawn of the ASN.1 standard in 1994, but with the following noteworthy advantage: the ObjectSet formalizes the information that was formerly (if ever) indicated in comments within the module (an example was given on page 346).

The INSTANCE OF type has the same tag 8 of class UNIVERSAL as the EXTERNAL type (a presentation context negotiation type presented in Section 14.1 on page 298). The BER encoding of an INSTANCE OF value is, therefore, compatible with the BER encoding of an EXTERNAL value that

[26]The context-specific tag [0] ensures compatibility with the EXTERNAL type (see Section 14.1 on page 298).

uses only the `syntax` and `data-value` components (see Section 18.2.19 on page 411).

The inter-personal e-mail standard X.420 provides an example (slightly adapted here) of use of the `TYPE-IDENTIFIER` class and its associated `INSTANCE OF` type to model the parameters and the values of a part of the message body whose type is not known in advance by the standard[27]:

```
ExtendedBodyPart ::= SEQUENCE {
  parameters [0] INSTANCE OF TYPE-IDENTIFIER OPTIONAL,
  data       INSTANCE OF TYPE-IDENTIFIER }
  (CONSTRAINED BY {-- must correspond to the &parameters --
   -- and &data fields of a member of -- IPMBodyPartTable})
```

15.9.2 Reference Manual

UsefulObjectClassReference → `TYPE-IDENTIFIER`
| `ABSTRACT-SYNTAX`

⟨1⟩ The `TYPE-IDENTIFIER` class is defined by:

```
TYPE-IDENTIFIER ::= CLASS {
  &id   OBJECT IDENTIFIER UNIQUE,
  &Type }
WITH SYNTAX { &Type IDENTIFIED BY id }
```

Type notation

InstanceOfType → `INSTANCE OF` *DefinedObjectClass*

⟨2⟩ This type has tag no. 8 of class `UNIVERSAL`. It has the same tag as the `EXTERNAL` type (see Section 14.1 on page 298).
⟨3⟩ *DefinedObjectClass* must be of class `TYPE-IDENTIFIER`.
⟨4⟩ To make value definition easier, the `INSTANCE OF` type is defined using the following associated type:

```
SEQUENCE {
  type-id DefinedObjectClass.&id,
  value   [0] DefinedObjectClass.&Type }
```

[27]The user-defined constraint introduced by the keywords `CONSTRAINED BY` is presented in Section 13.13 on page 294.

⟨5⟩ The only relevant constraint that can be applied to this type is a *SimpleTableConstraint* (see on page 349). In this case, a constraint like '`INSTANCE OF` *DefinedObjectClass* `({`*ObjectSet*`})`' is equivalent to :

```
SEQUENCE {
  type-id DefinedObjectClass.&id ({ObjectSet}),
  value   [0] EXPLICIT DefinedObjectClass.&Type
                          ({ObjectSet}{@.type-id}) }
```

that is, it contains a component of type `OBJECT IDENTIFIER` and a component of an open type (see rule ⟨3⟩ on page 347) whose value must be of the type determined by the object identifier.

Value notation

InstanceOfValue → *Value*

⟨6⟩ *Value* must conform to the `SEQUENCE` type associated with the `INSTANCE OF` type defined in rule ⟨4⟩ on the preceding page.

15.10 The pre-defined `ABSTRACT-SYNTAX` class

This class provides a means of linking items of information that are frequently used in specifications. It is standardized in [ISO8824-2, annexes A and B] and can, therefore, be used directly in a module like any other pre-defined ASN.1 type.

15.10.1 User's Guide

In Chapters 2 and 3, we have defined the protocol data units (PDU) of a specification. Every PDU defines an abstract syntax[28].

In Section 3.2 on page 20 (and more particularly on Figure 3.2 on page 22), we explained that, during the stage of negotiation between the Application and Presentation layers, the abstract syntax is referenced by an object identifier (see Section 10.8 on page 153). Even though it has hardly ever been used in specifications so far, the information object class `ABSTRACT-SYNTAX` can model this association of an object identifier to an abstract syntax or PDU (an ASN.1 type).

[28]If a specification defines more than one PDU, they can be gathered together in a `CHOICE` to constitute a single ASN.1 type, which stands for the abstract syntax.

It is defined by:

```
ABSTRACT-SYNTAX ::= CLASS {
  &id       OBJECT IDENTIFIER,
  &Type     ,
  &property BIT STRING {handles-invalid-encodings(0)}
              DEFAULT {} }
WITH SYNTAX { &Type IDENTIFIED BY &id
              [HAS PROPERTY &property] }
```

in which the `handles-invalid-encodings` property indicates that a decoder that cannot decode the received data does not treat them as errors but hands them over 'as is' to the receiving application, which will choose the more appropriate method to take them into account (in other words, it enables to indicate the Presentation layer not to interrupt the connection but notify the emitting application that an error has occurred when decoding).

The ASN.1 standard recommends grouping all the objects of class `ABSTRACT-SYNTAX` in a module that imports the PDUs from other modules:

```
ProtocolName-Abstract-Syntax-Module {iso member-body(2)
  f(250) type-org(1) ft(16) asn1-book(9) chapter15(3)
  protocol-name(0)}
DEFINITIONS ::= BEGIN
IMPORTS ProtocolName-PDU FROM ProtocolName-Module {iso
          member-body(2) f(250) type-org(1) ft(16)
          asn1-book(9) chapter15(3) protocol-name(0)
          module1(2)};
protocolName-Abstract-Syntax ABSTRACT-SYNTAX ::=
  {ProtocolName-PDU IDENTIFIED BY
     protocolName-Abstract-Syntax-id}
protocolName-Abstract-Syntax-id OBJECT IDENTIFIER ::=
  {iso member-body(2) f(250) type-org(1) ft(16)
   asn1-book(9) chapter15(3) protocol-name(0)
   abstract-syntax(0)}
protocolName-Abstract-Syntax-descriptor ObjectDescriptor
  ::= "Abstract syntax of ProtocolName"
protocolName-Transfer-Syntax-id OBJECT IDENTIFIER ::=
  {iso member-body(2) f(250) type-org(1) ft(16)
   asn1-book(9) chapter15(3) protocol-name(0)
   transfer-syntax(1)}
protocolName-Transfer-Syntax-descriptor ObjectDescriptor
  ::= "Transfer syntax of  ProtocolName"
END
```

Of course, the information contained in the `protocolName-Abstract-Syntax` object can be used by ASN.1 compilers (before 1994, this information was indicated in comments in the root module of the specification). Numerous examples of the `ABSTRACT-SYNTAX` class can be found in the X.500 directory recommendation series.

An information object of class `ABSTRACT-SYNTAX` is often parameterized (see Chapter 17) because constraint boundaries appearing in PDU(s) are fixed by some protocol specializations for example. In this case, every protocol specialization defines a (non-parameterized) `ABSTRACT-SYNTAX` class, which sets up the parameters of the 'generic abstract syntax'. If these subtyping boundaries are fixed only when implementing the protocol, they are called *parameters of the abstract syntax*; they will be discussed further in Section 17.3 on page 389.

The `ABSTRACT-SYNTAX` class is used in the type definition of the Presentation layer data [ISO8823-1]:

```
PDV-list ::= SEQUENCE {
  transfer-syntax-name
    Transfer-syntax-name OPTIONAL,
  presentation-context-identifier
    Presentation-context-identifier,
  presentation-data-values CHOICE {
    single-ASN1-type  [0] ABSTRACT-SYNTAX.&Type
      (CONSTRAINED BY {-- Type which corresponds to --
      -- the presentation context identifier --}),
    octet-aligned     [1] IMPLICIT OCTET STRING,
    arbitrary         [2] IMPLICIT BIT STRING }}
```

`presentation-context-identifier` is a number allocated to each presentation context: it is a unique integer for every presentation connection, which avoids the (sometimes expensive) transfer of an object identifier. The open type `ABSTRACT-SYNTAX.&Type` recalls that the `single-ASN1-type` alternative encodes values of an abstract syntax specified in ASN.1 and conforms to the negotiated presentation context identifier (the user-defined constraint introduced by `CONSTRAINED BY` was described in Section 13.13 on page 294).

15.10.2 Reference Manual

$$\textit{UsefulObjectClassReference} \rightarrow \underline{\texttt{TYPE-IDENTIFIER}}$$
$$\mid \texttt{ABSTRACT-SYNTAX}$$

⟨1⟩ The `ABSTRACT-SYNTAX` class is defined by:

```
ABSTRACT-SYNTAX ::= CLASS {
  &id  OBJECT IDENTIFIER,
  &Type,
  &property BIT STRING { handles-invalid-encodings(0) [a]}
               DEFAULT {} }
WITH SYNTAX { &Type IDENTIFIED BY &id
             [HAS PROPERTY &property] }
```

[a]This property indicates that invalid encodings are not treated as errors when decoding so that the decision is up to the application. In an OSI environment, this property prevents the Presentation layer to interrupt the connection by rejecting an invalid encoding.

⟨2⟩ The `ABSTRACT-SYNTAX` class was designed to reference types that are complete protocol data units or PDU (such a type is typically the `CHOICE` of the messages that the application can exchange); the PDU values are generally embedded in a value of type `EMBEDDED PDV` or `EXTERNAL` (see Chapter 14) of some lower-level protocol.

⟨3⟩ It is recommended that, in the context of a given specification, all the objects of class `ABSTRACT-SYNTAX` should be gathered in a dedicated module.

Chapter 16

Enough to read macros

Contents

They came to the conclusion that syntax was a fantasy and grammar an illusion. Besides, new rhetorics at the time announced that one should write as one speaks, and that all would be for the best provided one had experienced and observed.

Gustave Flaubert, *Bouvard and Pécuchet*.

If this chapter has not been entitled 'All about macros', it was quite deliberately so, for we do not aim at describing how to *write macros*. We would rather help the reader to understand macros that may crop up in various standards and thereby contribute to their updating to the ASN.1:1994/97 standard.

Indeed, the concept of macro itself is not mentioned in the ASN.1:1997 standard any more; it has been replaced since 1994 by the information object classes and information objects presented in Chapter 15. For these reasons, this chapter has no section called 'Reference Manual' that would have described exhaustively the notion.

16.1 Historical background

At the time ASN.1 still had the name 'Recommendation X.409', the remote operation service element (that would become later ROSE and be moved to the X.200 recommendation series) was an integral part of the X.409 recommendation. The ROSE working group designed the concept of `OPERATION` as a general mechanism for invoking the execution of an operation on a remote system and receive the result or an error message; this mechanism was introduced in the X.409 recommendation as a new type constructor.

In 1983, as explained in the history on page 61, the X.409 recommendation was being revised at ISO in view of its adoption. As the ISO working group did not fully grasp why a concept of operation would have to be included in an abstract notation, they refused this addition.

The opinions of the two organizations could have diverged and enden in two separate standard texts, one for CCITT and another for ISO, but CCITT did not accept the principle and proposed a general macro notation thus presented in [X.409, clause 4.3]: "*it is occasionally useful to define* non *standard type and/or value notation for a particular data type. Such notation is defined by means of a* macro. *[...] The body of a macro specifies the desired non-standard notation using BNF. Thus the body comprises a series of productions. The first production specifies the non-standard type notation, the second specifies the non-standard value-notation, and the remaining productions define any non-terminals introduced by the first two productions*".

16.2 Why macros?

A macro could, with its own notation, capture semantic information that would be impossible to specify in ASN.1 and provide the same functionality as the information objects do in a more formal way (see Chapter 15).

Using all the power of the Backus-Naur Form (BNF), a macro defines a new type notation and a new value notation, whereby they offer new degrees of liberty to the specifier since they allow the specifier to write, within an ASN.1 module, type or value definitions in any computing language's specific syntax for instance. The basic principles of ASN.1 remain unchanged: a type is a set of values; a value is still governed by a type. Besides, macros[1] do not extend types or values that can be defined in ASN.1 but offer instead a notation more appropriate to a specific application domain (like the X.400 email or the X.500 directory, for example).

A macro cannot be used to specify a PDU since it is impossible to access its internal elements but, as we shall see shortly, it enables to constrain ('parameterize') a PDU containing 'gaps' to be filled according to the specific needs of communicating applications.

16.3 General syntax of a macro

A macro is defined by two characteristics:

- a new type notation (as a constrained collection of one or several types that can be defined in ASN.1 without macros), and
- a new value notation for this type,

that is, two grammars, whatever their complexity, plus an ASN.1 definition, if needed, to specify the macro's semantics.

Since a macro extends the ASN.1 grammar, its name should respect the same lexical rules as ASN.1 keywords, i.e. consist only of capital letters[2].

A macro definition conforms to the framwork below, which uses the keywords `MACRO`, `BEGIN`, `END`, `TYPE NOTATION` and `VALUE NOTATION`:

```
MACRO-NAME MACRO ::=
BEGIN
TYPE NOTATION ::= -- type syntax --
VALUE NOTATION ::= -- value syntax --
-- grammatical productions used for defining
-- the type syntax and the value syntax
END
```

[1]In the rest of this chapter, we will talk about '*macro definition*' and '*macro instance definition*'.

[2]This convention has been kept for denoting information object classes (see Chapter 15).

If the macro notation includes the following syntactic entity (in which the reserved keywords are underlined)	**then the specifier should write at the same place in the macro instance definition**:
`"text"`	`text` (no double quotes)
`string`	any string made of the characters “`A`” to “`Z`”, “`a`” to “`z`”, “`0`” to “`9`”, “`:`”, “`=`”, “`,`”, “`{`”, “`}`”, “`<`”, “`.`”, “`(`”, “`)`”, “`[`”, “`]`”, “`-`”, “`'`”, “`"`”. N.B.: this character set is more restricted than that now used in ASN.1:1997 (see footnote 1 on page 100).
`identifier`	an identifier (word beginning with a lower-case letter)
`number`	an integer (non negative!)
`empty`	nothing
`type`	a type reference or the expression of an ASN.1 type (i.e. anything that may appear after the “`::=`” symbol in a type assignment)
`type(X)` where `X` is a typereference (word beginning with an upper-case letter, see on page 103)	an ASN.1 type called `X` in the macro instance
`value(type-expression)` (such a syntactic entity may appear several times in a macro definition)	an ASN.1 value conform to the ASN.1 type indicated in round brackets
`value(x X)` where `x` is a word beginning with a lower-case letter in X.208:1990 and by an upper-case letter in ISO 8824:1988, and where `X` is a typereference (word beginning with an upper-case letter, see on page 103)	an ASN.1 value conform to the ASN.1 type locally referenced by `X` in this macro instance; this value will be called `x` in the macro instance

.../...

Table 16.1: Syntactic entities used in macro definitions

If the macro notation includes the following syntactic entity (in which the reserved keywords are underlined)	**then the specifier should write at the same place in the macro instance definition**:
.../...	
`value(VALUE X)` (such a syntactic entity must appear only once per macro definition)	an ASN.1 value returned by the macro instance (e.g. when referencing the macro in an `ANY DEFINED BY` clause); if this value has to be encoded, it is the `X` type that must be taken into account
`<x type-expression ::= value-expression>`	definitions in angles "<" and ">" do not correspond to a syntax definition but associate semantics described in basic ASN.1 notation (i.e. without macros) with some elements of the macro; these are definitions of ASN.1 types or values that can be used in local references to the macro instance (such as `x` and `X` in the previous syntactic entities)
`<VALUE type-expression ::= value-expression>` appearing anywhere in the macro definition	embedded definition which gives ASN.1 semantics to the value returned by the macro instance: it is a value of type `type-expression` that is written as a `value-expression` (`type-expression` and `value-expression` use local references like `x` and `X` above)

Table 16.2: Syntactic entities used in macro definitions (continued)

A macro definition can be interpreted as a form to be filled in that includes directions depending on what has been inserted ("if the word XYZ was inserted then please go to rule Abc"). The grammatical productions have names that begin with an upper-case letter and they are separated by the "|" symbol. They may also use the particular keywords and syntactic entities[3] of Table 16.1 on page 366, that will be explained and illustrated further on.

16.4 First example: complex numbers

We propose a macro for defining complex number according to a notation that with is close to the mathematical formalism $(x + iy)$:

```
COMPLEX MACRO ::=
BEGIN
TYPE NOTATION ::= "Re" "=" type(ReType) ","
                  "Im" "=" type(ImType)
VALUE NOTATION ::= value(ReValue ReType) "+"
                   value(ImValue ImType) "i"
  <VALUE SEQUENCE { real-p      ReType,
                    imaginary-p ImType} ::=
         { real-p      ReValue,
           imaginary-p ImValue }>
END
```

After TYPE NOTATION, we find the new type notation provided by our COMPLEX macro: every instance of this macro should be denoted by the word Re, followed by the "=" symbol, and the ASN.1 description of the type of the real part of the complex number (this type is stored in the 'local variable' ReType), and after a comma, the word Im, the "=" symbol and the ASN.1 type description of the imaginary part (stored in the local variable ImType).

The VALUE NOTATION clause defines a new value notation offered by this macro: in every instance of this macro, we indicate, after the "::=" symbol, an ASN.1 value conform to the type ReType (this value is stored in the local variable ReValue), then the "+" symbol, followed by the imaginary part of the complex number as an ASN.1 value conform to the type ImType (this value is stored in the local variable ImValue) and we terminate by the "i" character.

[3]We call 'syntactic entity' a series a words or symbols that constitutes a semantic item in ASN.1.

The following example gives a complex number whose real and imaginary parts are integers:

```
c1 COMPLEX
     Re = INTEGER,
     Im = INTEGER ::=
  5 + 3 i
```

and the next illustrates the case where imaginary and real parts are ASN.1 real numbers[4]:

```
c2 COMPLEX
     Re = REAL,
     Im = REAL ::=
  {56,10,0} + {3561,10,-3} i
```

`c1` and `c2` are *value instances* of the `COMPLEX` macro.

If several complex numbers, with real and imaginary parts of `REAL` type for every one of them need to be defined, the type definition can be factorized[5]:

```
REAL-COMPLEX ::= COMPLEX
                   Re = REAL,
                   Im = REAL
```

Note that we have here the equivalent of a parameterized type, which could be written `COMPLEX{REAL,REAL}` if we used the parameterization introduced in ASN.1 in 1994 (see Chapter 17). Nevertheless, because this macro is not delimited (by the famous and popular curly brackets of ASN.1), it is difficult for a parser to handle it. `REAL-COMPLEX` is a *type instance* of the `COMPLEX` macro. We may therefore use `REAL-COMPLEX` as if it were some new ASN.1 primitive type:

```
c2 REAL-COMPLEX ::= {56,10,0} + {3561,10,-3} i
```

The definition in angles "<" and ">" indicates how the new type and value notations introduced with our macro should be interpreted in 'pure' ASN.1.

[4]Since macros have disappeared from the standard in 1994, we adopt here the obsolete ASN.1:1990 syntax of real values, i.e. without the identifiers `mantissa`, `base` and `exponent`.

[5]No example of macros was given in the ASN.1:1990 standard but it was only mentioned that a type notation defined by a macro could be used each time an ASN.1 type was expected and that a value notation based on a macro could appear each time a value was expected.

Hence the macro instance `c2` previously defined is equivalent to the definition:

```
c2 SEQUENCE { real-p      REAL,
              imaginary-p REAL } ::=
  { real-p      {56,10,0},
    imaginary-p {3561,10,-3} }
```

This embedded definition also indicates that, because of the keyword `VALUE`, the value returned by the macro instance `c2` is {`real-p {56,10,0}, imaginary-p {3561,10,-3}`}, that is to say if we define the (not-so-common!) type:

```
Addition ::= SEQUENCE {
  complex1 COMPLEX Re=INTEGER, Im=INTEGER,
  complex2 REAL-COMPLEX }
```

the value[6]:

```
result Addition ::= {
  complex1 5 + 3 i,
  complex2 {56,10,0} + {3561,10,-3} i}
```

is equivalent to:

```
result SEQUENCE {
    complex1 SEQUENCE { real-p      INTEGER,
                        imaginary-p INTEGER},
    complex2 SEQUENCE { real-p      REAL,
                        imaginary-p REAL}} ::= {
  complex1 { real-p      5,
             imaginary-p 3 },
  complex2 { real-p      {56,10,0},
             imaginary-p {3561,10,-3} }}
```

The type specified in the embedded definition in angles is the one that is going to be used when encoding the value returned by the macro. In our case, the value returned by instances of the `COMPLEX` macro are encoded according to the type `SEQUENCE {real-p REAL, imaginary-p REAL}`.

Unfortunately, an embedded definition containing the keyword `VALUE` does not systematically appear in all macro definitions (it is even the case for macro definitions that appear in standard texts!) and it is up

[6]Note that is not possible to associate the previously defined values `c1` and `c2` with the identifiers `complex1` and `complex2` because the expected values must respect the value notation of the `COMPLEX` macro.

to the specifier to understand the hidden semantics of the macro to be instanced. Moreover, the absence of an embedded definition between angles makes it impossible for ASN.1 compilers to deal with macros in all their generality.

16.5 Second example: the macro `OPERATION` of ROSE

Macros have historically resulted from the definition of the Remote Operation Service Element ROSE [ISO9072-2] presented on page 80. The first macro defined in this standard was the `OPERATION` macro for which every instance defines information related to an operation that should be remotely executed: the argument type, the result type, the linked operations if any, errors to be returned and its identification number.

As we shall see, the `OPERATION` macro definition[7] calls for many syntactic entities defined in Table 16.1 on page 366. It does not contain, however, any definition in angles "<" and ">"; the value returned by such a macro instance is either an integer (local identification) or an object identifier (global identification):

```
OPERATION MACRO ::=
BEGIN
TYPE NOTATION ::= Argument Result Errors
                  LinkedOperations
VALUE NOTATION ::= value (VALUE
  CHOICE { localValue  INTEGER,
           globalValue OBJECT IDENTIFIER })

Argument ::= "ARGUMENT" NamedType | empty
Result ::= "RESULT" ResultType | empty
Errors ::= "ERRORS" "{" ErrorNames "}" | empty
LinkedOperations ::=
    "LINKED" "{" LinkedOperationNames "}"
  | empty
ResultType ::= NamedType | empty
NamedType ::= identifier type | type
ErrorNames ::= ErrorList | empty
ErrorList ::= Error | ErrorList "," Error
Error ::= value(ERROR) | type
LinkedOperationNames ::= OperationList | empty
```

[7]This definition calls another macro named `ERROR` whose definition is not reproduced here.

```
OperationList ::= Operation
                | OperationList "," Operation
Operation ::= value(OPERATION) | type
END
```

The operation that returns a subscriber's name according to its phone number could be specified by the following macro instance:

```
get-subscriber-name OPERATION
 ARGUMENT NumericString (SIZE (10))
 RESULT   IA5String
 ERRORS   {unknown, db-unavailable}
  ::= localValue:1
```

where we have used the alternative `empty` of the production `LinkedOperations` since there is no linked operation. Remember that a macro is quite like a form with gaps to fill in: if the production `Argument` is being applied, then we write the word '`ARGUMENT`', and go to the production `NamedType`...

As mentioned already, a macro and its instances collect information that 'pure' ASN.1 cannot represent, but it cannot be encoded. If an application is asked to execute an operation, we use the following type whose values will contain the code[8] of an operation and its argument:

```
Invoke ::= SEQUENCE {
 opcode   OPERATION,
 argument ANY DEFINED BY opcode }
```

The component `argument` is of the type indicated after the word '`ARGUMENT`' in the macro instance that returns the value `opcode`[9]. As a result, for the `get-subscriber-name` instance that returns the value `localValue:1`, the `argument` component has the type `NumericString (SIZE (10))`.

This last sentence "*whose component `argument` is of the type indicated after the word '`ARGUMENT`' in the macro instance which returns the value `opcode`*" is obviously all but formal in the `Invoke` type (at best is it indicated in comments or in a textual part of the standard) and it is,

[8]It is actually such an abusive use of an `opcode` component of type `CHOICE` (the type of the values returned by the `OPERATION` macro) in the `ANY DEFINED BY` type that has lead the standard designers to tolerate this practice since 1990 (see History on page 63) and to generalize it in ASN.1:1994 (see rule ⟨7⟩ on page 318).

[9]The type of the component `opcode` is the one indicated after the keyword `VALUE` in the macro definition, which means that for the BER encoding, this type adopts either the tag of class `UNIVERSAL` of the `INTEGER` type or that of the `OBJECT IDENTIFIER` type.

therefore, impossible to make use of such links automatically in an ASN.1 compiler unless these were implemented by hand.

The inconvenience of the `ANY DEFINED BY` type was already mentioned in Section 12.8 on page 241. In particular, we have shown that an association table represented by the `ANY DEFINED BY` type could be indicated at the beginning of the module. The other possibility for representing this table is a list of macro instances defined in the module. Still, nothing formally specifies whether all these instances or only some of them can be used. The information object sets introduced in 1994 and presented in Section 15.5 on page 329 solve the problem and can be directly treated by compilers.

16.6 Main (and major!) disadvantages of macros

In addition to the `ANY` type, macro instances are one of the historical mistakes of ASN.1, which was corrected in 1994. Their numerous disadvantages induced many specifiers to use them abusively or erroneously. At worst did they refuse to use ASN.1, forgetting many of its strong points.

First, macros provide all the flexibility offered by the BNF notation and potentially allow the specifiers to rewrite all the ASN.1 grammar without making clear what they are allowed to do or where not to venture lest it produces macro definitions that cannot be parsed by compilers. Moreover, the standard text was itself ambiguous and included bugs, which were the subject of numerous defect reports. [Ste93] gives the following macro (conform to ASN.1:1988):

```
VORACIOUS MACRO ::=
BEGIN
TYPE NOTATION ::= Eat
VALUE NOTATION ::= empty
Eat ::= "END" | "MACRO" | EatSomething Eat
EatSomething ::= type | identifier | number
               | Keyword | Special
Keyword ::= "NULL" | "TRUE" | "FALSE"
          | "PLUS-INFINITY" | "MINUS-INFINITY"
Special ::= "::=" | "," | "{" | "}" | "." | "(" | ")"
  | "’" string "’B" | "’" string "’H" | """" string """"
END
```

whose instances have the disastrous behavior of consuming whatever is found until the next macro definition (or the end of the module)!

Second, macro definitions did not have to appear before their instances. If a compiler was bound to treat them in their full generality, the parser had to be dynamically extensible[10] so that new grammar rules can be added to it. But, since macro instances were not delimited (not even by the so very popular curly brackets, which can be found anywhere else in ASN.1!), it proved impossible to treat them automatically if the macro definitions did not appear systematically *before* their use.

Besides, unlimited syntactic elements prevent from producing efficient parsers since it makes error recovery impossible [ASU86]. Many compilers, therefore, supported the most frequently used macro definitions (in particular, those of ROSE [ISO9072-2]), which were directly implemented within the parser[11].

Finally, words used in macros (such as `ARGUMENT`, `RESULT`...) had no formal semantics since the macro specifier was free to write a completely new grammar. As a consequence, it was impossible to use these informal semantic links when automatically generating encoders and decoders. The only link was restricted to the `ANY DEFINED BY` type, as formally exposed. This link was still restricted at the level of only one `SEQUENCE` or `SET` type, and it could not be set between distinct parts of a given PDU (all this has been possible since 1994 as explained in Section 15.8 on page 352).

16.7 Macro substitutes since 1994

Depending on the context where it is used, a macro definition should not systematically be replaced by an information object class.

If a macro aimed rather at providing a particular form (as in Figure 15.1 on page 311) that the communicating application designer would fill in with information specific to its application domain (see

[10]Refer on this point to [Rin95] who produced a parser by a clever use of functional programming. The few restrictions imposed by the author on macro usage clearly show that these cannot be supported by any ASN.1 compiler in their generality.

[11]Even though they bear the same name, ASN.1 macros have nothing in common with macros of the *C* language, which only define a textual substitution operated by the lexical analyzer using a pre-processor and impose no changes to the parser (see Figure 22.1 on page 464).

the `OPERATION` macro of ROSE on page 371 or the `ATTRIBUTE` information object class of the X.500 directory on page 327), it is replaced by an *information object class definition* (see Chapter 15) with which is associated a *user-friendly syntax* (introduced by the keywords `WITH SYNTAX`) accordingly defined.

The translation of macro instances into information objects is merely a matter of nesting the macro instance body in curly brackets, slightly shifting the "`::=`" symbol and changing the comma for a vertical bar "`|`" in some lists (see migration of ROSE macros in [ISO13712-1, annex C]); the information objects can then be grouped in an object set (generally dynamically extensible).

If the macro is the equivalent of a parameterized type like the `SIGNED` macro of recommendation [X.509]:

```
SIGNED MACRO ::=
BEGIN
  TYPE NOTATION ::= type(ToBeSigned)
  VALUE NOTATION ::= value(VALUE SEQUENCE {
    ToBeSigned,
    AlgorithmIdentifier, -- of the algorithm used
      -- to generate the signature
    ENCRYPTED OCTET STRING -- where the octet string
      -- is the result of the hashing of the value of
      -- "ToBeSigned" -- })
END
```

it should be replaced by a parameterized type (or a parameterized value, see Chapter 17):

```
SIGNED{ToBeSigned} ::= SEQUENCE {
  toBeSigned    ToBeSigned,
  COMPONENTS OF SIGNATURE{ToBeSigned} }
```

If the macro specifies constraints that cannot be formalized in ASN.1 like the `ENCRYPTED` macro of recommendation [X.509]:

```
ENCRYPTED MACRO ::=
BEGIN
  TYPE NOTATION ::= type(ToBeEnciphered)
  VALUE NOTATION ::= value (VALUE BIT STRING
    -- the value of the bit string is generated by
    -- taking the octets which form the complete
    -- encoding (using the ASN.1 BER) of the value
    -- of the ToBeEnciphered type and applying an
    -- encipherment procedure to those octets --)
END
```

it is replaced by a user-defined constraint introduced by the keywords `CONSTRAINED BY` (see Section 13.13 on page 294).

The annex defining the macro notation was permanently removed from the ASN.1 standard in its 1997 edition.

Chapter 17

Parameterization

Contents

Abstraction itself cannot be separated from generalization.

Armand Cuvillier, *Dictionnary of philosophical language.*

The various entities that can be specified in ASN.1 have been exposed in the previous chapters of this Reference Manual. We now have to show how our specifications can be improved by being made more compact and more flexible. We also demonstrate in which respect parameters can be excellent substitutes for macros and `ANY` types, which are no longer part of the standard since 1994.

17.1 Basics of parameterization

In ASN.1, parameters are equivalent to parameters of functions in computing languages[1]. And similarly as a computing program can be generalized by parameterizing a function to apply it to different arguments, an

[1]Or equivalent to bound variables of λ-calculus for those into a more theoretical approach...

ASN.1 specification can be generalized by parameterizing assignments to be applied to different actual parameters. All ASN.1 concepts (types, values, value sets, information object classes, objects and object sets) can be parameterized and the parameter itself can be one of those six concepts.

Several reasons may incline a specifier to parameterize some definitions.

First, the specification can have several structurally identical definitions, for example, the two following types, which model coordinates in a plan:

```
Pair1 ::= SEQUENCE { x INTEGER,
                     y INTEGER }
Pair2 ::= SEQUENCE { x REAL,
                     y REAL }
```

In such a case, the specifier can group them in a single definition, which is parameterized by the coordinates' type to make the specification more compact and easily generalizable.

Second, the specifier can leave gaps in the generic specification because it will be re-used and generalized by other working groups. Hence parameters can advantageously replace erroneous uses of the now-obsolete `ANY` type (or even abusive use of the `OCTET STRING` type in some RFCs of the Internet). The parameter value is fixed in a *functional profile*[2] or in an *international standardized profile*[3].

Finally, some specifications may include parameters specific to particular implementations (frequently for subtype constraint boundaries)

[2]Some very generic standards sometimes offer a great variety of options (choice between several service elements, parameters...). But it is difficult to dictate that the implementations of these standards should include all the options that can be combined in very different ways. A group of application designers, an industrial consortium, a regional organization or a national standardization organization can therefore produce a *functional profile* to restrict this variety.

[3]An *international standardized profile* (ISP) is a functional profile produced at an international level by ISO. It can harmonize local functional profiles to improve interworking between different implementations. Before standardization, an ISP should go through all the states of an international standard (see on page 55 and following).

and it is always annoying to limit a standard or an international recommendation to today's technical restrictions. Such parameters are documented in a *protocol instance compliance statement*[4] (PICS). These are called parameters of the abstract syntax.

Parameters have been standardized since 1994 in Part 4 of the ASN.1 standard [ISO8824-4].

Before going further, we define a few vocabulary points: we call *formal parameter*, a 'gap' left by the specifier in a definition; it is equivalent to the expressions *parameter* and *dummy reference* in the standard text. Similarly, an *actual parameter* is an instance of a formal parameter; it is usually called *argument* in programming languages.

17.2 Parameters and parameterized assignments

17.2.1 User's Guide

The list of formal parameters is denoted in curly brackets (but how could it be otherwise?!) right after the assignment name; the parameters are then used on the right-hand part (after the "`::=`" symbol) of the definition as if they were references of types, values, etc.

The [X.520] directory service defines a type where the maximum length of each character string type is parameterized as follows:

```
DirectoryString{INTEGER:maxSize} ::= CHOICE {
  teletexString   TeletexString (SIZE (1..maxSize)),
  printableString PrintableString (SIZE (1..maxSize)),
  universalString UniversalString (SIZE (1..maxSize)),
  bmpString       BMPString (SIZE (1..maxSize)),
  utf8String      UTF8String (SIZE (1..maxSize)) }
```

The parameter `maxSize` begins with a lower-case letter and is *governed* by the `INTEGER` type, so it represents a value of this type.

To determine the category of a parameter from its name and its governor if it exists, we can refer to Table 17.1 on the next page. When compared with Table 15.1 on page 314, it shows that the rules are similar

[4]A *protocol instance compliance statement* (PICS) details in a table the choices made by a manufacturer for a particular implementation on its specific equipment. We find there additive functionalities compared to those imposed by the protocol specification as well as the variation interval of the parameters of the abstract syntax.

If the governor is	and if the parameter name	then the parameter is
absent	begins with an upper-case letter	a type
a type	begins with an lower-case letter	a value
a type	begins with an upper-case letter	a value set
absent	is entirely in upper-case letters	a class (or a type)
a class name	begins with an lower-case letter	an object
a class name	begins with an upper-case letter	an object set

Table 17.1: The different categories of parameters and governors

to those which determine the category of an information object class field.

To use[5] the parameterized type `DirectoryString` previously defined, we attribute a value to its `maxSize` parameter:

```
SubstringAssertion ::= SEQUENCE OF CHOICE {
  initial [0] DirectoryString{ub-match},
  any     [1] DirectoryString{ub-match},
  final   [2] DirectoryString{ub-match} }
ub-match INTEGER ::= 128
```

`ub-match` is an actual parameter since it appears only on the right-hand side of the `SubstringAssertion` assignment.

The application of the actual parameter `ub-match` to the parameterized type `DirectoryString` amounts to substituting `ub-match` for all the occurrences of the actual parameter `maxSize`, which implies that the expression `DirectoryString{ub-match}` is strictly equivalent to:

```
CHOICE {
  teletexString   TeletexString (SIZE (1..ub-match)),
  printableString PrintableString (SIZE (1..ub-match)),
  universalString UniversalString (SIZE (1..ub-match)),
  bmpString       BMPString (SIZE (1..ub-match)),
  utf8String      UTF8String (SIZE (1..ub-match)) }
```

If the `SubstringAssertion` type should be made generic, it needs to be parameterized too:

```
SubstringAssertion{INTEGER:ub-match} ::= SEQUENCE OF
  CHOICE { initial [0] DirectoryString{ub-match},
           any     [1] DirectoryString{ub-match},
           final   [2] DirectoryString{ub-match} }
```

[5]A parameterized type is only taken into account by a compiler to produce the associated encoding and decoding functions if all its parameters are allocated.

`ub-match` is then a formal parameter since it appears at the left-hand side of the assignment. This formal parameter is 'propagated' to the `DirectoryString` type as an actual parameter (instance of its formal parameter `maxSize`).

The scope of a formal parameter is limited to the right-hand side of the definition where it occurs so that it is possible (and even recommended when parameters are propagated) to use the same parameter name in several assignments. Moreover, within its scope a parameter takes precedence over any other reference that would have the same name; for the following example, the type `T` is not taken into account by the parameterized `List` type (the parameter `T` wins over):

```
T ::= INTEGER
List{T} ::= SEQUENCE OF T
```

If a parameter that is a type appears in a `SEQUENCE`, `SET` or `CHOICE` type where distinct tags are required (see rules ⟨14⟩ on page 224, ⟨10⟩ on page 228 and ⟨9⟩ on page 238), it will have to be tagged by the specifier unless the module includes the `AUTOMATIC TAGS` clause in its header:

```
Choice{T} ::= CHOICE { a [0] T,
                       b INTEGER }
Structure{T} ::= SEQUENCE { a INTEGER,
                            b [0] T OPTIONAL,
                            c INTEGER }
```

Indeed, if such a parameter is assigned, any ASN.1 type can potentially be substituted to it (and therefore any tag). We find here again the notion of open type mentioned on page 315 when extracting a type field from an information object class.

For the same reason, a parameter that is a type should necessarily be tagged in `EXPLICIT` mode (see rule ⟨6⟩ on page 216) so that its instance tag could be encoded and the decoder could decode the value according to its type (in BER).

A parameter that is a type can be the governor of a parameter that is a value or a value set. This enables parameterizing the default type and default value of a component of a `SEQUENCE` or `SET` type, as in:

```
GeneralForm{T, T:val} ::= SEQUENCE {
  info     T DEFAULT val,
  comments IA5String }
```

A specialisation of this type could be written:

```
Form ::= GeneralForm{BOOLEAN, TRUE}
```

which is equivalent to the type:

```
Form ::= SEQUENCE { info     BOOLEAN DEFAULT TRUE,
                    comments IA5String }
```

In the same manner as what we did for a type, a parameterized value is given by:

```
pariTierce{INTEGER:first, INTEGER:second,
       INTEGER:third} SEQUENCE OF INTEGER ::=
  { first, second, third }
```

When a definition includes several parameters, it can prove better to use a single parameter given by an information object (see Chapter 15) that collects all the parameters[6]:

```
MESSAGE-PARAMETERS ::= CLASS {
  &max-priority-level        INTEGER,
  &max-message-buffer-size   INTEGER,
  &max-reference-buffer-size INTEGER }
WITH SYNTAX {
  MAXIMUM PRIORITY &max-priority-level
  MAXIMUM MESSAGE BUFFER &max-message-buffer-size
  MAXIMUM REFERENCE BUFFER &max-reference-buffer-size }

Message-PDU{MESSAGE-PARAMETERS:param} ::= SEQUENCE {
  priority INTEGER (0..param.&max-priority-level
                   !Exception:priority),
  message  UTF8String (SIZE
             (0..param.&max-message-buffer-size)
             !Exception:message),
  comments UTF8String (SIZE
             (0..param.&max-reference-buffer-size)
             !Exception:comments) }
Exception ::= ENUMERATED {priority(0), message(1),
                          comments(2), ...}
```

Numerous examples[7] of parameterized references are given in [ISO8824-4, annex A]. The last edition of the ROSE standard

[6]The exception marker "`!`" is defined on page 247.

[7]In our section about subtype constraints introduced by the keywords `CONSTRAINED BY`, we also presented an example for these parameters (see on page 295).

[ISO13712-1] constitues an excellent real-world example of use of parameters from which the following two examples are extracted:

```
Forward{OPERATION:OperationSet} OPERATION ::=
  { OperationSet |
    OperationSet.&Linked.&Linked |
    OperationSet.&Linked.&Linked.&Linked.&Linked }
Reverse{OPERATION:OperationSet} OPERATION ::=
  { Forward{{OperationSet.&Linked}} }
```

`Forward` is an object set of class `OPERATION` parameterized by an object set of the same class and is made up by the union of three object sets `OperationSet`, `OperationSet.&Linked.&Linked` and `OperationSet.&Linked.&Linked.&Linked.&Linked` (see Section 15.5 on page 329). `Reverse` is an object set of class `OPERATION` also parameterized by an object set of the same class. Note the two levels of curly brackets for the actual parameter `Forward`: the outermost level indicates that it is a parameter, the innermost level indicates that it is an object set made by extracting the `&Linked` information from the object set `OperationSet`.

Finally, when importing or exporting parameterized definitions in the module header, it is recommended to follow each parameterized reference up with a pair of curly brackets in the `EXPORTS` and `IMPORTS` clauses. For the last two types, it gives:

```
ModuleName DEFINITIONS ::=
BEGIN
EXPORTS Forward{}, Reverse{}, ForwardAndReverse;
IMPORTS
  OPERATION FROM Remote-Operations-Information-Objects
      {joint-iso-itu-t remote-operations(4)
       informationObjects(5) version1(0)}
  Forward{}, Reverse{}
    FROM Remote-Operations-Useful-Definitions
      {joint-iso-itu-t remote-operations(4)
       useful-definitions(7) version1(0)};
-- dynamically extensible object set:
MyOperationSet OPERATION ::= {...}
-- non-parameterized definition:
ForwardAndReverse OPERATION ::=
  {Forward{{MyOperationSet}} UNION Reverse{{MyOperationSet}}}
END
```

17.2.2 Reference Manual

ParameterizedAssignment →
  ParameterizedTypeAssignment
  | *ParameterizedValueAssignment*
  | *ParameterizedValueSetTypeAssignment*
  | *ParameterizedObjectClassAssignment*
  | *ParameterizedObjectAssignment*
  | *ParameterizedObjectSetAssignment*

⟨1⟩ The right-hand part of a *ParameterizedAssignment* (i.e. after "`::=`") cannot consist only in a formal parameter *DummyReference*.
⟨2⟩ Normally an abstract syntax should not contain formal parameters on its PDU level, except if these parameters appear in a subtype constraint (production *Constraint* on page 293), in which case they are called *parameters of the abstract syntax* and the constraint is said to be *variable*. If the actual parameters are not defined in the abstract syntax, the values implemented can be specified in a protocol instance compliance statement (PICS, see footnote 4 on page 379). When a parameter remains declared as a variable in a PICS, it may take different values from one communication instance to another, and even change during a given communication instance.

ParameterizedTypeAssignment →
  typereference *ParameterList* "`::=`" *Type*

⟨3⟩ A *ParameterizedTypeAssignment* must not contain a direct or indirect reference to itself unless such a reference is directly or indirectly marked `OPTIONAL` in a structure or unless this reference is included in a `CHOICE` type where one of the alternatives is not circular.
⟨4⟩ In a *ParameterizedTypeAssignment*, a formal parameter *DummyReference* that is a type cannot be used as a tagged actual parameter within a recursive reference to this *ParameterizedTypeAssignment* (an example that justifies this rule is given in [ISO8824-4, clause A.3]).

ParameterizedValueAssignment →
  valuereference *ParameterList* *Type* "`::=`" *Value*

⟨5⟩ A *ParameterizedValueAssignment* must not contain a direct or indirect reference to itself.

ParameterizedValueSetTypeAssignment →
  typereference *ParameterList* *Type* "`::=`" *ValueSet*

⟨6⟩ A *ParameterizedValueSetTypeAssignment* must not contain a direct or indirect reference to itself unless such a reference is directly or indirectly marked `OPTIONAL` in a structure or unless this reference is included in a `CHOICE` type where one of the alternatives is not circular.
⟨7⟩ In a *ParameterizedValueSetTypeAssignment*, a formal parameter *DummyReference* that is a type cannot be used as a tagged actual parameter within a recursive reference to this *ParameterizedValueSetTypeAssignment* (an example that justifies this rule is given in [ISO8824-4, clause A.3]).

ParameterizedObjectClassAssignment →
objectclassreference *ParameterList* "::=" *ObjectClass*

⟨8⟩ A *ParameterizedObjectClassAssignment* must not contain a direct or indirect reference to itself unless such a reference is directly or indirectly marked `OPTIONAL`.

ParameterizedObjectAssignment →
objectreference *ParameterList* *DefinedObjectClass* "::=" *Object*

⟨9⟩ A *ParameterizedObjectAssignment* must not contain a direct or indirect reference to itself.

ParameterizedObjectSetAssignment → objectsetreference
ParameterList *DefinedObjectClass* "::=" *ObjectSet*

⟨10⟩ A *ParameterizedObjectSetAssignment* must not contain a direct or indirect reference to itself.

ParameterList → "{" *Parameter* "," $\cdots^{+}$ "}"
Parameter → *ParamGovernor* ":" *DummyReference*
| *DummyReference*

⟨11⟩ If there is an ambiguity on the syntax of a formal parameter *DummyReference* (is it an information object class or a type? Is it an object or a value?), it can generally be removed on the first occurrence of the formal parameter at the right-hand side of the definition. If the formal parameter is used as an actual parameter of a *ParameterizedReference* (see on page 117), its nature should be recursively determined by the definition of this *ParameterizedReference*.

ParamGovernor → *Governor*
| *DummyGovernor*
Governor → *Type*
| *DefinedObjectClass*

DummyGovernor → *DummyReference*
DummyReference → *Reference*

⟨12⟩ The governor and the governed parameter must respect the rules of the semantic model of ASN.1 presented in Section 9.4 on page 121.
⟨13⟩ If the formal parameter *DummyReference* is a value or a value set, the governor *ParamGovernor* must be present and it must be a type (or a value set) that restricts the set of possible values.
⟨14⟩ If a formal parameter *DummyReference* is a value or a value set, all the governor's values must be valid for all the occurrences of the parameter at the right-hand side of the parameterized reference.
⟨15⟩ If the formal parameter *DummyReference* is an information object or an information object set, the governor *ParamGovernor* must be present and it must be the name of an information object class (in particular, the governor of an information object set cannot be another object set).
⟨16⟩ If the formal parameter *DummyReference* is a type or an information object class, it must have no governor *ParamGovernor*.
⟨17⟩ If the formal parameter *DummyReference* is a type and if it is used where distinct tags are required (in a `CHOICE`, a `SET` or in a group of optional components of a `SEQUENCE`, for instance), it must be tagged in `EXPLICIT` mode by the specifier (see rule ⟨6⟩ on page 216) unless the module includes the `AUTOMATIC TAGS` clause in its header. Examples of tagging are given in [ISO8824-4, clause 9.8].
⟨18⟩ The governor of a formal parameter *DummyReference* must not contain a reference to another formal parameter *DummyReference* if the latter is governed.
⟨19⟩ The governor of a formal parameter *DummyReference* should require neither the knowledge of the formal parameter nor that of the parameterized reference being defined.
⟨20⟩ The scope of a formal parameter *DummyReference* is the *ParameterList* itself, and the right-hand part (after the symbol "`::=`") of the *ParameterizedAssignment*.
⟨21⟩ Every formal parameter *DummyReference* must be used at least once within its scope (see the preceding rule).
⟨22⟩ A formal parameter *DummyReference* takes precedence over all the other references of the same name appearing in its scope.
⟨23⟩ The use of a formal parameter *DummyReference* in its scope must be consistent with its syntax and, if need be, with its governor, as well as with any other occurrences of this *DummyReference*. Sometimes, the

consistency checking needs to be delayed until the parameter is instantiated, such as in the following example where the type `Color` depends on what the value `blue` is:

```
Flag{Color} ::= SEQUENCE {
  country VisibleString,
  colors  SEQUENCE OF Color DEFAULT {blue} }
```

Reference → typereference | valuereference
| objectclassreference | objectreference
| objectsetreference

⟨24⟩ *Reference* is here a local variable name and not a reference to an entity (type, value, object...) defined in the module.

Reference to a parameterized type or to a parameterized value set

ParameterizedType → *SimpleDefinedType ActualParameterList*
ParameterizedValueSetType →
SimpleDefinedType ActualParameterList

SimpleDefinedType → *ExternalTypeReference*
| typereference

⟨25⟩ typereference must be the name of a *ParameterizedTypeAssignment* or a *ParameterizedValueSetTypeAssignment* of the current module, or must appear as a *Symbol* in the `IMPORTS` clause of the current module.
⟨26⟩ The production *ExternalTypeReference* is presented on page 118.

Reference to a parameterized value

ParameterizedValue → *SimpleDefinedValue ActualParameterList*
SimpleDefinedValue → *ExternalValueReference*
| valuereference

⟨27⟩ valuereference must be the name of a *ParameterizedValueAssignment*, or must appear as a *Symbol* in the `IMPORTS` clause of the current module.
⟨28⟩ The production *ExternalValueReference* is presented on page 119.

Reference to a parameterized information object class

ParameterizedObjectClass →
  DefinedObjectClass *ActualParameterList*

⟨29⟩ *DefinedObjectClass* (see on page 119) must be the name of a *ParameterizedObjectClassAssignment* of the current module, or must appear as a *Symbol* in the `IMPORTS` clause of the current module.

Reference to a parameterized information object

ParameterizedObject → *DefinedObject* *ActualParameterList*

⟨30⟩ *DefinedObject* (see on page 120) must be the name of a *ParameterizedObjectAssignment* of the current module, or must appear as a *Symbol* in the `IMPORTS` clause of the current module.

Reference to a parameterized information object set

ParameterizedObjectSet →
  DefinedObjectSet *ActualParameterList*

⟨31⟩ *DefinedObjectSet* (see on page 120) must be the name of a *ParameterizedObjectSetAssignment*, or must appear as a *Symbol* in the `IMPORTS` clause of the current module.

Common productions

ActualParameterList → "{" *ActualParameter* "," $\cdots^{+}$ "}"

ActualParameter → *Type* | *Value*
  | *ValueSet* | *DefinedObjectClass*
  | *Object* | *ObjectSet*

⟨32⟩ There must be exactly one *ActualParameter* for each formal parameter appearing in the corresponding *ParameterizedAssignment* (see on page 384) and they must appear in the same order.
⟨33⟩ The meaning of a reference appearing as an *ActualParameter* and the global tagging mode (see Section 12.1.3 on page 213) for this actual parameter (if it denotes a type) depend on the actual parameter's tagging environment and not on the corresponding formal parameter's.
⟨34⟩ If the *ActualParameter* is a value set or an object set, it must be nested in curly brackets not to be confused with a type (and because the grammar productions *ValueSet* and *ObjectSet* contain curly brackets).

17.3 Parameters of the abstract syntax

As described in the previous section, parameters can be propagated from one parameterized definition to another, but this chain of parameters stop sooner or later on the specification's top level type: the PDU, which constitutes what is called the *abstract syntax* of messages exchanged between communicating applications (see Section 3.2 on page 20).

But every now and then, a specifier may propagate a parameter, which corresponds, for example, to a subtype constraint boundary; there is no reason why such a parameter should be set up in a standardized specification whereas it would rather depend on technical characteristics of the implementation.

A parameter that remains formal at the PDU level because it depends on the implementation is called *parameter of the abstract syntax*. It is allowed only if it appears in subtype constraints (and can therefore not be used in the component of a `SEQUENCE` type, for instance).

The parameters of the abstract syntax are assigned in an international standardized profile (ISP, see footnote 3 on page 378) if they correspond to the specialization of an existing standard or in a protocol instance compliance statement (PICS, see footnote 4 on page 379) if they depend on a particular implementation.

But the fact that parameters are different for every implementation is prone to interworking errors. As a consequence, it is recommended to use an exception marker[8] "`!`" for subtype constraints that include parameters of the abstract syntax:

```
CharacterString{INTEGER:max-length} ::= CHOICE {
  teletexString   TeletexString (SIZE (1..max-length)
                               !exceeds-max-length),
  printableString PrintableString (SIZE (1..max-length)
                                 !exceeds-max-length) }
exceeds-max-length INTEGER ::= 999
```

If the decoder receives oversized data (because the sender has not used the same value for the parameter of the abstract syntax `max-length`), it triggers the exception `exceeds-max-length` which enables calling a dedicated procedure for these data and, for example, warn the user.

[8]The use of an exception marker in subtype constraints was exposed on page 292; it corresponds to the production *ExceptionSpec* presented on page 255.

In Section 15.10 on page 359, we recommended defining an information object of class ABSTRACT-SYNTAX in a module that contains a protocol PDU to clearly identify it and associate its object identifier. If this PDU includes parameters of the abstract syntax, these (but no others) should appear as parameters of the information object of class ABSTRACT-SYNTAX:

```
my-abstract-syntax {INTEGER:maxSize} ABSTRACT-SYNTAX ::=
  { my-PDU{size-max} IDENTIFIED BY {iso
    member-body(2) f(250) type-org(1) ft(16)
    asn1-book(9) chapter17(4) my-PDU(0)} }
```

If the parameters of the abstract syntax are used at a low level in the PDU, they should be propagated through all the definitions that may require them. Indeed, it is impossible to define global parameters to make them known by all the definitions of a module, for instance[9].

We have already come across a special case of parameters of the abstract syntax that were not necessarily associated with the top-level: an object set that contains an extension marker "..." is dynamically extensible (see on page 346). The communicating application can add or remove objects during the connection. It is not necessary to propagate such a parameter from the PDU down to the level of the definitions that use it, since this property is implicit for every occurrence of the extensible object set.

[9]The concept of global parameter was examined by the ASN.1 working group (document ISO/IEC JTC 1/SC 21 N 9734), but was finally discarded since it appeared that few users would actually need it.

Part III

Encoding Rules and Transfer Syntaxes

Chapter 18

Basic Encoding Rules (BER)

Contents

In three words I can sum up everything I've learned about life. It goes on.

Robert Frost.

This third part of the book is focused on the standard encoding and decoding rules associated with ASN.1. It can be ignored by specifiers who, generally, may not be concerned with the way the data they describe is encoded. They can go directly to Chapter 22 where the practical use of these encoding rules are discussed from the ASN.1 compiler viewpoint.

Before proceeding to the following three chapters, the reader may refer again to Chapters 2 and 3 to brush up the notions of abstract syntax, transfer syntax and encoding rules, which are crucial for this part. Remember in particular that applying encoding rules to an abstract syntax provides a transfer syntax (see Figure 2.4 on page 15).

The Basic Encoding Rules, or BER, are historically the original encoding rules of ASN.1 since they were already part of the [X.409] standard before it was split up into two parts in 1985 (see Section 6.3.2 on page 61). The term 'basic' indicated that other rules might be standardized in the future; it actually happened in 1994 when the packed encoding rules (PER) presented in Chapter 20 were introduced in the standard.

18.1 Main principles

All the rules defined in the current chapter and the next two are presented from the encoding standpoint and should obviously be interpreted the other way around to obtain the decoding rules. Besides, we clearly assume that the ASN.1 specification to be used is semantically correct, i.e. all the rules enounced in Part II 'Reference Manual' are respected.

The BER transfer syntax always has the format of a triplet TLV ⟨Type, Length, Value⟩ (often noted ⟨Tag, Length, Value⟩ too) as in Figure 18.1(a) on the next page. All the fields T, L and V are series of octets. The value V can, itself, be a triplet TLV if it is *constructed* (see Figure 18.1(b) on the next page). The most complex of the ASN.1 values are no more than a stack of less and less complex values.

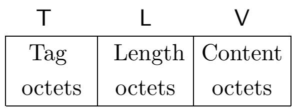
The transfer syntax is octet-based[1] and self-delimited since the field L provides a means of determining the length of each TLV triplet. Because the BER are 'big Endians' (see on page 8), the high-order bit is at the left-hand side; it is numbered 7 as in Figure 18.1(c) on the next page.

The T*ag* octets (but generally one octet is enough) correspond to the encoding of the tag of the value's type. One of the bits specifies the form (basic or constructed) of the V content octets. These tag octets identify, therefore, the value unambiguously in its context[2]. They should conform to one of the two formats of Figure 18.2 on page 396.

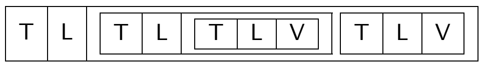
If the tag number is smaller than or equal to 30, the tag class and number are encoded on a single octet as shown at the bottom of Figure 18.2 (all tags of class `UNIVERSAL` are grouped in Table 12.1 on page 209). If the tag number is greater than 30, the second form is used; then the number consists of the concatenation of the bits from no. 6 down to

[1]To avoid confusion with any internal representation mode (as already explained in Section 10.7 on page 151), we use the word *octet* instead of *byte* throughout the third part of the book.

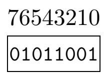
[2]This context is still rule-governed by the condition for distinct tags (see rule ⟨11⟩ on page 217).



(a) Triplet TLV



(b) Recursive principle



(c) Bit weights

Figure 18.1: BER transfer syntax (TLV format)

no. 0 for all octets but the first one, whose five lower-order bits equal 11111. The bottom of Figure 18.2 on the next page shows that for all octets but the last one, the bit no. 7 equals 1. If the tag is not encoded on a number of bits that is a multiple of 7, some of the bits from no. 1 to 6 of the second octet are unused and filled with padding 0s. In these two forms, the bit no. 5 of the first octet indicates whether the value (i.e. the V field) is encoded in basic or constructed form. These two forms are used in the next section.

The length octets represent the length of the value that is actually encoded, i.e. the number of content octets used in the V part of the TLV triplet. If the bit no. 5 of the first tag octet indicates a primitive encoding form (see Figure 18.2 on the following page), the length is encoded in definite form. If the bit indicates a constructed encoding form, the sender may choose to code the length in definite or indefinite form.

As shown on Figure 18.3 on the next page, the *definite form* can be short (if the length of the V field is shorter than 127) or long, depending on the sender. This liberty allows, for example, the protocol layer to encode all the length fields on a fixed number of octets, for specific needs of two communicating systems. If the sender should not be allowed to choose the form of the length field, the CER or DER encoding rules, derived from BER, can be used (see Table 19.1 on page 420).

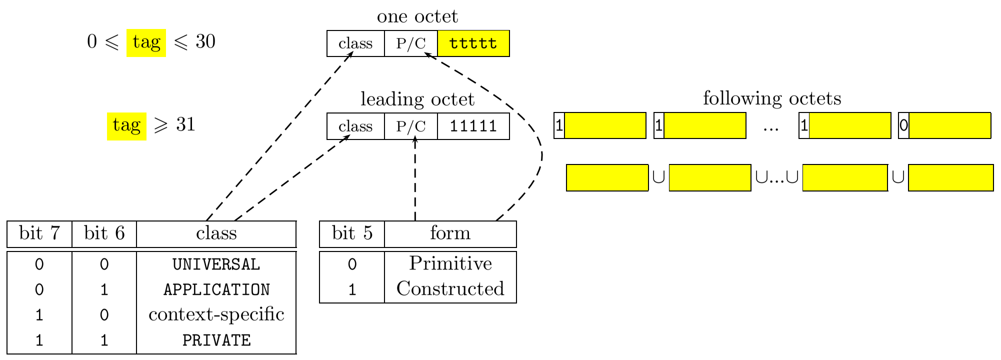


Figure 18.2: The two formats of the tag octets (T)

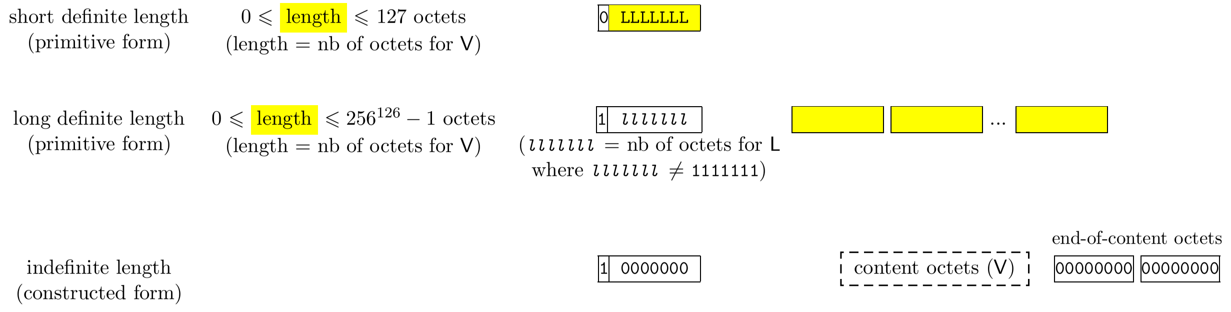

short definite length (primitive form)
$0 \leqslant$ length $\leqslant 127$ octets (length = nb of octets for V)
0 LLLLLLL
long definite length (primitive form)
$0 \leqslant$ length $\leqslant 256^{126} - 1$ octets (length = nb of octets for V)
1 lllllll
(lllllll = nb of octets for L where lllllll ≠ 1111111)
...
indefinite length (constructed form)
1 0000000
content octets (V)
end-of-content octets
00000000
00000000

Figure 18.3: The three formats of the length octets (L)

In the long form, the first octet of the L part represents the length of the length, i.e. the number of octets necessary for encoding the length (it cannot be encoded on 127 octets, for this length is reserved for future extensions).

The encoding of the *indefinite form* is more specifically used when the whole content part is not available for the sender so that its length cannot be computed before sending the whole TLV triplet. For example, if the Application layer 'cuts up' and hands over the data to the Presentation layer by small items and if the Presentation layer has little memory space available, it may start transmitting without knowing the whole data length. The encoding in indefinite form also prevents double-scanning the values (once for computing their length and a second time to actually encode them) and therefore provides more efficient encoders. Unfortunately, the decoder, on the other side, cannot allocate the adequate memory space before receiving all the data. Note that splitting the data into small items like this is not allowed in DER encoding (see Table 19.1 on page 420).

If the value is encoded in indefinite form, two zero octets are added after the encoding of the value (which is necessarily in constructed form). These two trailing octets are in fact a TLV triplet that stands for the encoding of a value tagged `[UNIVERSAL 0]` of zero length (hence there is no V octet in the content field). This justifies a posteriori our keeping the tag no. 0 of class `UNIVERSAL` in Table 12.1 on page 209 and ensures an unambiguous encoding since no ASN.1 type can ever use this tag.

Generally speaking, the T tag field and L length field extend the V value of only two octets. The V content field is discussed in the next section.

The BER encoding rules are registered with the object identifier {`joint-iso-itu-t(2) asn1(1) base-encoding(1)`} in the registration tree of Figure 10.4 on page 161 and the object descriptor `"Basic Encoding of a single ASN.1 type"` (see Section 11.15 on page 198). This object identifier will be used in a negotiation phase as presented on Figure 3.2 on page 22 to indicate that the BER transfer syntax must be associated with the abstract syntax proposed by the communicating application.

18.2 Encoding of all types

In this section, the examples of encoding are (almost) systematically provided with a tag of class `UNIVERSAL` for the field T and a value encoded in the short form. Other encoding forms can of course be chosen by the sender whenever these are allowed and the T field can encode other classes of tags in case of tagging in implicit mode.

18.2.1 `BOOLEAN` value

The encoding of the boolean values is necessarily primitive. The `FALSE` value is encoded as a triplet[3]:

T	L	V
1_{10}	1_{10}	00000000

The value `TRUE` is encoded on any single not-null octet; the sending application may choose, for example:

T	L	V		T	L	V
1_{10}	1_{10}	11111111	or	1_{10}	1_{10}	10110011

18.2.2 `NULL` value

The `NULL` value is encoded without value octet:

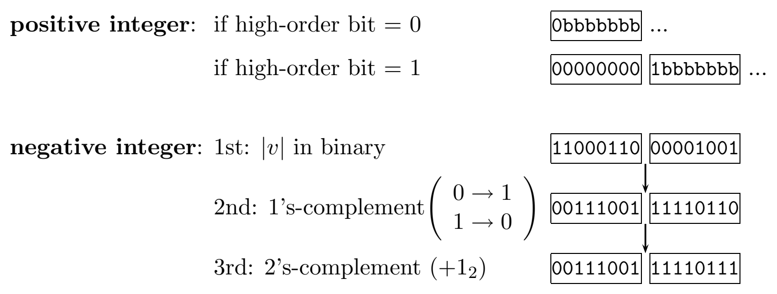
T	L	V
5_{10}	0_{10}	

18.2.3 `INTEGER` value

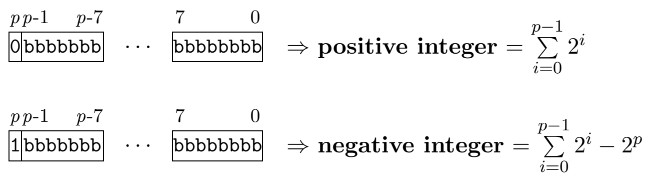
The encoding of the `INTEGER` values is necessarily in primitive form. According to the encoding/decoding diagrams on Figure 18.4 on the next page, the content octets represent the binary encoding of the integer if it is positive, or its two's-complement if it is negative[4]. The first octet equals zero when a positive integer is encoded on a whole number of octets whose high-order bit is set to `1`. If the number of bits is not a

[3] In the rest of this chapter, the subscript notation n_{10} denotes a number n in base 10 (not in base 2) for reading's sake.

[4] The only null integer is `0`; there exists no `-0` integer as in the one's-complement arithmetic.


positive integer: if high-order bit = 0
0bbbbbbb ...
if high-order bit = 1
00000000 1bbbbbbb ...
negative integer: 1st: |v| in binary
11000110 00001001
2nd: 1's-complement (0 → 1, 1 → 0)
00111001 11110110
3rd: 2's-complement (+1₂)
00111001 11110111

(a) encoding


p p-1 p-7
0bbbbbbb
7 0
bbbbbbbb
⇒ positive integer = Σ_{i=0}^{p−1} 2^i
p p-1 p-7
1bbbbbbb
7 0
bbbbbbbb
⇒ negative integer = Σ_{i=0}^{p−1} 2^i − 2^p

(b) decoding

Figure 18.4: Two's-complement for integers

multiple of 8, the high-order bits are filled with the adequate number of `0`s before two's-complementing it if necessary. All the first nine bits cannot equal `0` or `1`.

Using Figure 18.4, we find, for example, that the integer -27,066 is encoded as:

$$\overset{\mathsf{T}=2}{\boxed{\mathtt{00}\,|\,\mathtt{0}\,|\,\mathtt{00010}}}\quad \overset{\mathsf{L}=2}{\boxed{\mathtt{00000010}}}\quad \overbrace{\boxed{\mathtt{10010110}}\;\boxed{\mathtt{01000110}}}^{\mathsf{V}=\text{-27,066}}$$

Nothing forbids the compiler from allocating a fixed-size memory space to store an integer whose type is constrained by an interval in the ASN.1 specification (see Section 13.4 on page 263).

18.2.4 `ENUMERATED` value

The integer associated with an enumeration identifier (which can be computed according to the rule ⟨12⟩ on page 140) is encoded following the directions given in the previous section. If the `ENUMERATED` type includes an extension marker, an identifier in the extensions is encoded as if it were part of the root because the BER implicitly take into account extensibility since there exists no restriction regarding the size of integers to be encoded.

18.2.5 `REAL` value

Even though, from the abstract syntax's viewpoint, the `REAL` type is semantically equivalent to a `SEQUENCE` type (see rule ⟨2⟩ on page 144), a value of that type is always encoded in primitive form according to one of the numerous possibilities presented in Figure 18.5 on the next page (others may be standardized in the future). Encoding rules like CER or DER (see Chapter 19) may restrict these possibilities. What follows ensures that ASN.1 `REAL` values in base `2` or `10` are properly decoded by the receiver (see rule ⟨8⟩ on page 144).

If the `base` component of the ASN.1 `REAL` value equals `10`, the character-based encoding is retained: the real number is represented as a string like '`123E+100`' according to the NR1, NR2 or NR3 formats defined in the [ISO6093] standard (see on page 143). This string is then encoded in ASCII (as a string of type `IA5String`). These formats may begin with unlimited blank spaces, and begin or end with unlimited zeros. The leading plus sign "`+`" is optional and the decimal separator is either a dot or a comma... (see Table 19.1 on page 420 for restricting these options).

If the `base` component of the ASN.1 `REAL` value equals `2`, the binary encoding is retained; the encoding base `BB` is chosen by the sender, generally according to its internal implementation of real numbers (it bears no relation with the `base` component of the abstract syntax). As computing systems do not generally store the mantissa as an integer, the scale factor `FF` may shift the data of one or two bits in base 8 or 16 to make up for the fact that the exponent can only shift them from 3 or 4 bits. This scale factor has the advantage of moving the decimal separator (dot or comma) on an octet's boundary, and prevent the encoder from shifting the mantissa to bring back its least significant bit on the octet's boundary (see [ISO8825-1, annex C]).

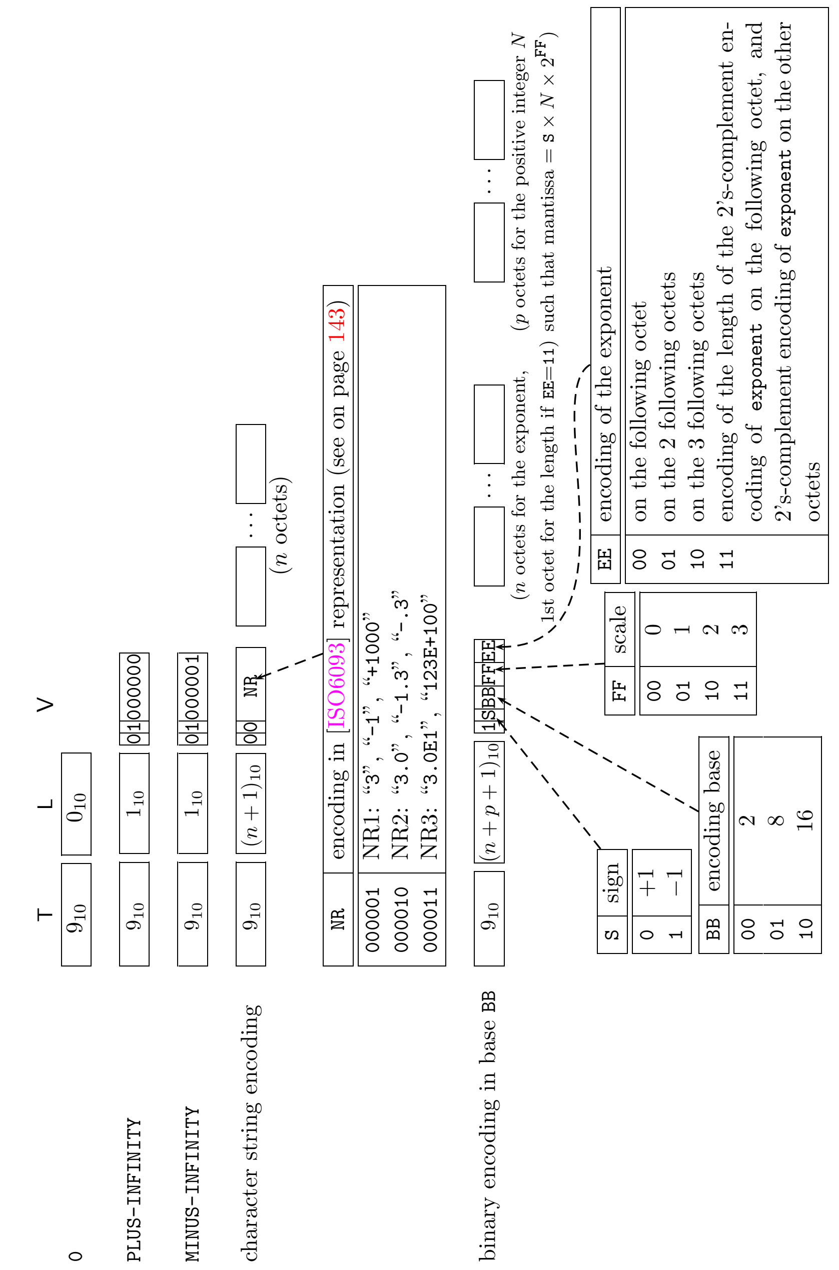

T
L
V
0
9_{10}
0_{10}
PLUS-INFINITY
9_{10}
1_{10}
01000000
MINUS-INFINITY
9_{10}
1_{10}
01000001
character string encoding
9_{10}
$(n+1)_{10}$
00
NR
···
(n octets)
NR | encoding in [ISO6093] representation (see on page 143)
000001 | NR1: "3", "-1", "+1000"
000010 | NR2: "3.0", "-1.3", "-.3"
000011 | NR3: "3.0E1", "123E+100"
binary encoding in base BB
9_{10}
$(n+p+1)_{10}$
1
S
BB
FF
EE
···
(n octets for the exponent,
1st octet for the length if EE=11)
···
(p octets for the positive integer N
such that mantissa = S $\times N \times 2^{\text{FF}}$)
S | sign
0 | +1
1 | −1
BB | encoding base
00 | 2
01 | 8
10 | 16
FF | scale
00 | 0
01 | 1
10 | 2
11 | 3
EE | encoding of the exponent
00 | on the following octet
01 | on the 2 following octets
10 | on the 3 following octets
11 | encoding of the length of the 2's-complement encoding of exponent on the following octet, and 2's-complement encoding of exponent on the other octets

Figure 18.5: The encoding forms of a REAL value

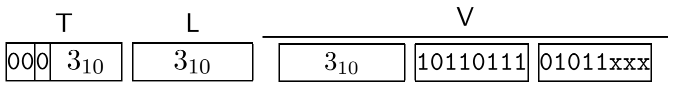
The binary encoding format is closer to the way the floating-point processors work. It allows the encoder to directly copy (*dump*) the mantissa's octets in their internal-memory storing format even if they are not positioned as they should be for transfer. Contrary to the `INTEGER` type, the mantissa is not encoded in two's-complement but going from one format to another is straightforward if the mantissa is known since the sign is given by the `S` bit.

The encoded real number is therefore equivalent to $\mathtt{S} \times N \times 2^{\mathtt{F}} \times \mathtt{BB}^{\mathtt{EE}}$ where $\mathtt{S} \times N \times 2^{\mathtt{F}}$, `BB` and `EE` are unrelated to the `mantissa`, `base` and `exponent` components of the ASN.1 abstract value.

Though slightly different from the usual formats handled by floating-point processors, the bi-directional conversion between the BER formats and the machine format for real numbers can be obtained very easily because the format adopted for BER is half-way between the various formats handled by computers. The drawback is that an encoder has to know all those different formats. We will see in the next chapter that the distinguished encoding rules (DER), for example, impose a single format (see Table 19.1 on page 420).

18.2.6 `BIT STRING` value

The encoding form can be primitive or constructed. In the primitive form, the string is cut up in octets and a leading octet is added so that the number of bits left unused at the end could be indicated by an integer between 0 and 7. The bit string `'1011011101011'B` is therefore encoded as:

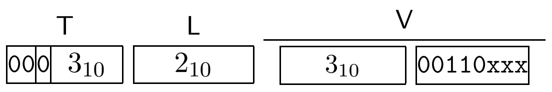


The unused bits are represented above by `xxx`, the sender can choose to pad them by `0`s or `1`s (possibly mixed).

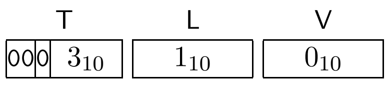
If the `BIT STRING` type includes a named-bit list in the specification, the decoder is free to add or remove a certain number of `0` bits at the end (see rule ⟨15⟩ on page 150).

```
Rights ::= BIT STRING { user-read(0), user-write(1),
                        group-read(2), group-write(3),
                        other-read(4), other-write(5) }
group Rights ::= { group-read, group-write }
```

This `group` value can therefore be encoded as:

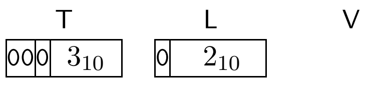


If the bit string is empty (or if it has no `1`-bits), the leading octet is set to `0` and may be followed (as a sender's option) by one or more other `0` octets. Very often, the only V octet present is the leading octet, which indicates the number of meaningless ending bits, that is:



The constructed form should be preferred if part of the data must be transmitted before the whole bit string is available. In this case, the bit no. 5 of the T tag octet equals `1`, the L octet indicates that the length is undefined (see Figure 18.3 on page 396) and the content octets are made of a series of TLV triplets where T necessarily[5] encodes the tag `[UNIVERSAL 3]` and where V represents a part of the bit string encoded in primitive or (recursively) constructed form.

Each part should have a length that is a multiple of 8 bits (hence its first value octet is necessarily `0`) except sometimes the last part for which the first value octet indicates the number of unused bits in the last value octet (padding with `0`s or `1`s is up to the sender). Empty parts of nil length can be inserted in the following form:

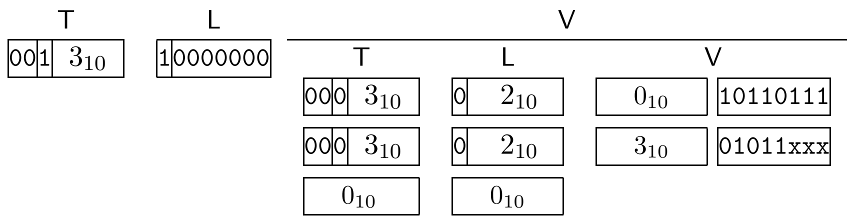


The octet stream ends with two null 'end-of-content' octets[6]. When receiving the data, the decoder builds up again the bit string by concatenating the content octets in the right order.

[5]This tag identification is redundant but it ensures the encoding homogeneity by preserving the TLV format in all levels of the BER transfer syntax.

[6]These two octets cannot be considered as belonging to a bit string because the length of each part is known.

If we go back on the binary string `'1011011101011'B`, an encoding in constructed form could be:



18.2.7 `OCTET STRING` value

The encoding of an octet string is similar to that of a `BIT STRING` value. As a consequence, the encoding can be in primitive or constructed form. Since there exists no unused bits at the end of the string, the leading octet does not appear (ASN.1 imposes that an octet string in binary or hexadecimal form should be filled with a sufficient number of `0`s so that its length could be a multiple of 8 bits, see rules ⟨4⟩ and ⟨6⟩ on page 153).

18.2.8 `OBJECT IDENTIFIER` value

The encoding is always in primitive form. It is the number associated with each arc that is encoded but not the identifiers. Every integer is encoded on a series of octets where all the bits no. 7 but for the last octet equals `1`. The encoding of the integer is therefore the concatenation of the bits from no. 6 down to no. 0 for each octet (this form is similar to the long-form encoding of the tag octets presented in Figure 18.2 on page 396).

The first two arcs of the registration tree (see Figure 10.4 on page 161) are encoded on a single integer using the formula[7] '$first\_arc \times 40 + second\_arc$', which assumes[8] that there is no more than 39 arcs just below the `iso` and `itu-t` arcs of the root. No limitation, however, is imposed on the number of arcs under `joint-iso-itu-t`.

[7]This formula legitimizes the rule ⟨4⟩ on page 166, which imposes that an object identifier must contain at least two arcs. One can get round this rule using the new `RELATIVE-OID` type (see Section 10.9 on page 167).

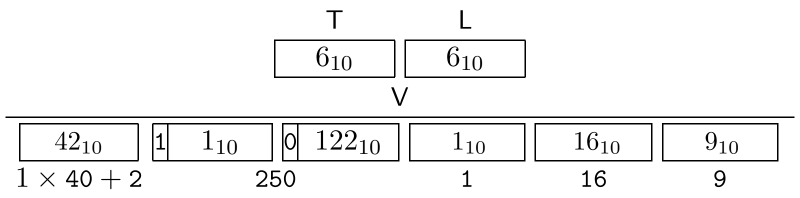
[8]Note that if the [ISO9834-1] standard for management of the international registration tree inserts a new arc below the root, the BER standard should be adapted as appropriate (while keeping the encoding compatibility) since such an addition is not provided for at the moment.

first octet(s)	**first arc**	**second arc**
$0 \leqslant n \leqslant 39$	`itu-t`	n
$40 \leqslant n \leqslant 79$	`iso`	$n - 40$
$n \geqslant 80$	`joint-iso-itu-t`	$n - 80$

Table 18.1: Decoding of the first octets of an object identifier

The decoding of the first content octet should therefore be carried out according to Table 18.1. Nevertheless, the decoder does not always need to decode an object identifier: it may only concatenate the content octets and compare the resulting string with those associated with the various syntaxes negotiated by the Presentation layer (in the case of a value embedded by one of the presentation context negotiation types of Chapter 14).

The encoding of the object identifier {`iso member-body f(250) type-org(1) ft(16) asn1-book(9)`} gives:

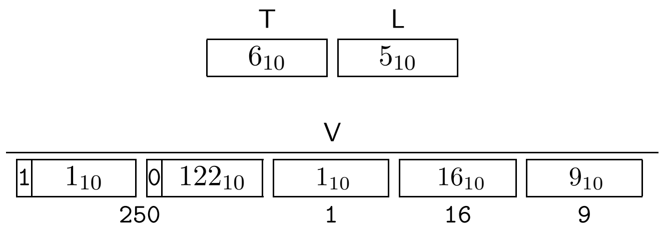


18.2.9 `RELATIVE-OID` value

The encoding is always in primitive form. A relative object identifier is encoded as an object identifier according to the rules stated in the previous section, that is to say that the integer associated with each arc is encoded on a series of octets where every no. 7 bit, but for the last octet, equals `1` (the encoding of the integer is therefore the concatenation of the bits from no. 6 down to 0 for each octet).

Contrary to the `OBJECT IDENTIFIER` type, there is no point in dealing separately with the first two arcs of the registration tree because the reference node of a relative object identifier cannot be the root nor a node just below this root (see rule ⟨3⟩ on page 169).

So the encoding of the relative object identifier {`f(250) type-org(1) ft(16) asn1-book(9)`} is given by:



18.2.10 Character strings and dates

All the character strings and dates that conform to the types presented in Chapter 11 are encoded as if they were octet strings of type `[UNIVERSAL` *`t`*`] IMPLICIT OCTET STRING` where *`t`* is the tag of class `UNIVERSAL` of the considered character string type (see Table 11.1 on page 175).

The strings of type `NumericString`, `PrintableString`, `IA5String`, `TeletexString` (or `T61String`), `VideotexString`, `VisibleString` (or `ISO646String`), `GraphicString`, `GeneralString`, `ObjectDescriptor`, `UTCTime` and `GeneralizedTime` are encoded on 8 bits according to the [ISO2022] standard and may use escape sequences[9] and character encodings registered according to the [ISO2375] standard. For each character string type, the character sets G, C0 and C1 (see on page 177) are fixed by default. These are not discussed here; they can be found in [ISO8825-1, Table 3].

Strings of type `UniversalString`, `BMPString` and `UTF8String` are encoded according to the [ISO10646-1] standard[10]. Every character is encoded on 4 octets for `UniversalString`, 2 octets for `BMPString` and the smallest number of octets conforming to [ISO10646-1Amd2] or [RFC2279] (see Table 11.3 on page 191) for `UTF8String`.

18.2.11 `SEQUENCE` value

The encoding is in constructed form. Every component is encoded as a TLV triplet; the ordering of the triplets is the same as the order of the

[9]Table 11.1 on page 175 describes the character string types that include escape characters.

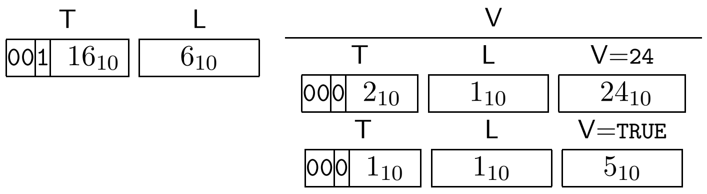
[10]Signatures must not be used. Control functions C0 and C1 must conform to the recommendations of [ISO8825-1, clause 8.20.9].

components in the `SEQUENCE` type definition in the ASN.1 specification (once developed, if present, the `COMPONENTS OF` operators according to rule ⟨19⟩ and following on page 224). A component marked `DEFAULT` is not necessarily encoded, even if the sending application provided a value for this component (it is a sender's option).

For example, the value:

```
v SEQUENCE { age    INTEGER,
             single BOOLEAN } ::= { age    24,
                                    single TRUE }
```

is encoded as:



Although the T tag octets are only necessary when encoding optional components (see rule ⟨14⟩ on page 224), they all need to be encoded to preserve homogeneity in the transfer syntax.

If the `SEQUENCE` type includes an extension marker, it is ignored and the components in the extensions are encoded as if they belonged to the extension root because the BER implicitly take extensibility into account thanks to tagging.

18.2.12 `SET` value

The principle of encoding `SET` values is the same as the `SEQUENCE` type, but the encoding ordering of the components is up to the sender; this is not necessarily the ordering of the ASN.1 specification (for this reason, the `SET` type has tag `17` of class `UNIVERSAL` whereas the `SEQUENCE` type has tag `16`). Besides, the decoder does not have to keep this order when providing the values to the receiving application.

If the `SET` type includes an extension marker, it is ignored and the components in the extensions are encoded as if they belonged to the extension root because the BER implicitly take extensibility into account thanks to tagging.

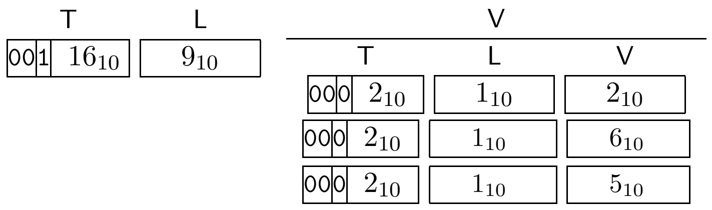
18.2.13 `SEQUENCE OF` value

The `SEQUENCE OF` type has the same tag `[UNIVERSAL 16]` as the `SEQUENCE` type. Thus it adopts the same encoding rules, i.e. every element of the list is encoded as if it were a component of a `SEQUENCE` type (so the tag field T of the encoding of every single element contains the redundant `UNIVERSAL` class tag of the type of these elements).

The encoding of the value:

```
triplet SEQUENCE OF INTEGER ::= {2, 6, 5}
```

is given by:



18.2.14 `SET OF` value

A value of type `SET OF` is encoded as if every one of its elements were the component of a `SET` type. The encoding ordering is chosen by the sender.

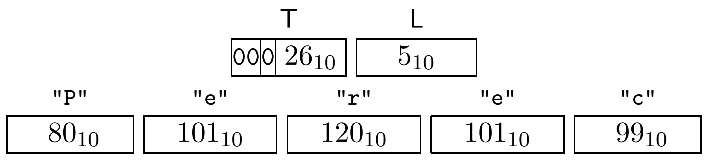
18.2.15 `CHOICE` value

Strictly speaking, a `CHOICE` type has no real existence but is only a way of proposing an alternative on the ASN.1 specification's level. A value of type `CHOICE` is encoded according to the type (and the tag) of the alternative that has been retained. The restrictions on distinct tags imposed by rule ⟨9⟩ on page 238 ensure the non-ambiguity of the decoding.

The encoding of the value:

```
famous CHOICE { name   VisibleString,
                nobody NULL } ::= name:"Perec"
```

is given by:



If the `CHOICE` type is preceded by a tag (which is necessarily of class `EXPLICIT`), like, for example, in:

```
famous [0] CHOICE { name   VisibleString,
                    nobody NULL } ::= name:"Perec"
```

the tag `[0]` is encoded in constructed form as explained in the following section.

If the `CHOICE` type includes an extension marker, it is ignored and an extension's alternative is encoded as if it belonged to the extension root because the BER implicitly take extensibility into account thanks to tagging.

18.2.16 Tagged value

If a type is tagged in `IMPLICIT` mode (or if the module includes the `IMPLICIT TAGS` or `AUTOMATIC TAGS` clause in its header)[11]:

```
v [1] IMPLICIT INTEGER ::= -38
```

only the tag[12] that appears on the left-hand side of the `IMPLICIT` keyword is encoded in the tag field T:

T			L	V
10	0	1_{10}	1_{10}	11011010

If the type is tagged in `EXPLICIT` mode (or if the module includes the `EXPLICIT TAGS` clause in its header):

```
v [APPLICATION 0] EXPLICIT INTEGER ::= 38
```

the value is encoded in constructed form as a series of TLV triplets where the tag fields T contain all the subsequent tags until the `UNIVERSAL` class tag of the type is encountered; this tag must be included in the encoding (see rules ⟨2⟩ and ⟨3⟩ on page 216):

[APPL. 0]			L	[UNIV. 2]			L	V
01	1	0_{10}	3_{10}	00	0	2_{10}	1_{10}	00100110

[11]Remember that `CHOICE` and `ANY` types, open types and parameters that are types cannot be tagged in `IMPLICIT` mode (see rule ⟨6⟩ on page 216).

[12]ASN.1 allows several tags to appear on the left-hand side of the `IMPLICIT` keyword (see rules ⟨3⟩ and ⟨4⟩ on page 216), but it is useless in practice. In such a case, the tags are encoded in constructed form as if there were an explicit tagging mode.

1994/97 version	1990 version	
alternative `identification` (see on page 301)	component `direct-reference` (see on page 299)	component `indirect-reference` (see on page 299)
`syntax`	`syntax`	ABSENT
`presentation-context-id`	ABSENT	`presentation-context-id`
`context-negotiation`	`transfer-syntax`	`presentation-context-id`

Table 18.2: Correspondence between abstract syntax and transfer syntax for the `EXTERNAL` type

18.2.17 Subtype constraints

Because the basic encoding rules were introduced before the subtype constraints in the ASN.1 standard, the latter were not supported by the BER. The length field `L` is, therefore, always transmitted even though the length is fixed by a `SIZE` constraint in the abstract syntax.

The subtype constraints' extensibility is implicitly taken into account by the BER.

18.2.18 `EXTERNAL` value

As discussed in Section 14.1 on page 298, the `EXTERNAL` type definition has changed between the 1990 and 1994/97 editions of the ASN.1 standard. To ensure upward compatibility of encodings, values of `EXTERNAL` type, even if they conform to the `SEQUENCE` type of Figure 14.1 on page 301 (1994 version), are encoded as if they conformed to the `SEQUENCE` type on page 299 (1990 version); the context-specific class tags, in particular, which appear before the alternatives of the `encoding` component (of type `CHOICE`) must be encoded but those computed in the 1994 version must not. The correspondence between the two versions is obtained in Table 18.2.

The value embedded in the value of type `EXTERNAL` is encoded according to the encoding rules denoted by the alternative `identification` (1994/97 version) and it is transmitted in one of the alternatives of the `encoding` component (1990 version): if the embedded value conforms to an ASN.1 type and if it is encoded according to the same encoding rules as the value of `EXTERNAL` type, any of the three alternatives can be retained; if the embedded value is encoded according to the encoding rules that were negotiated (i.e. agreed on by the two applications) and if it includes a whole number of octets, one of the two alternatives, `octet-aligned` or `arbitrary`, must be chosen; the `arbitrary` alternative is retained by default.

18.2.19 `INSTANCE OF` value

Since the `INSTANCE OF` type has the same tag of class `UNIVERSAL` as the `EXTERNAL` type, their encodings are identical. Besides, a value of type `INSTANCE OF` must conform to the `SEQUENCE` type of rule ⟨4⟩ on page 358. Comparing this type to the `SEQUENCE` type (1990 version) associated with the `EXTERNAL` type and defined on page 299 shows that the `type-id` component (`value` component respectively) of an `INSTANCE OF` value is transmitted in the `direct-reference` component (`single-ASN1-type` component respectively) of the `EXTERNAL` type. The encoding rules used for encoding the value embedded in the `value` component are the same as those used in the rest of the specification.

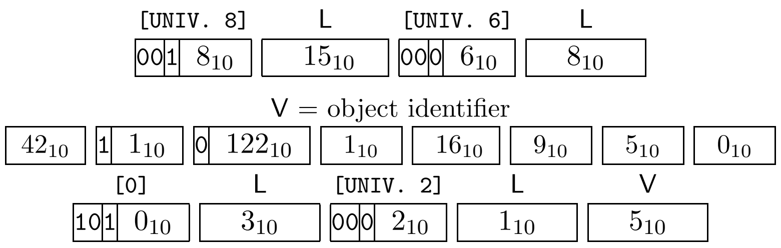
Since the `value` component is tagged in explicit mode, the BER encoding of the embedded value includes the tag of class `UNIVERSAL` of its ASN.1 type, which means that the value:

```
v INSTANCE OF TYPE-IDENTIFIER ::=
  { type-id {iso member-body f(250) type-org(1) ft(16)
             asn1-book(9) chapter18(5) integer-type(0)},
    value   INTEGER:5 }
```

is encoded as the equivalent value of the following `SEQUENCE` type:

```
{ direct-reference {iso member-body f(250) type-org(1)
                    ft(16) asn1-book(9) chapter18(5)
                    integer-type(0)},
  encoding         single-ASN1-type:INTEGER:5 }
```

that is:



For decoding, the distinction `EXTERNAL`/`INSTANCE OF` is no cause for misinterpretation for the receiving application: the object identifier of the `direct-reference` component denotes an abstract syntax for an `EXTERNAL` type or an ASN.1 type for an `INSTANCE OF` type.

18.2.20 `EMBEDDED PDV` value

An `EMBEDDED PDV` value is encoded according to the `SEQUENCE` type associated with the `EMBEDDED PDV` type and defined in Figure 14.3 on page 305. The embedded value is encoded[13] with respect to the transfer syntax identified by the `identification` component, and then transmitted by the `data-value` component of type `OCTET STRING`.

18.2.21 `CHARACTER STRING` value

A `CHARACTER STRING` value is encoded according to the `SEQUENCE` type associated with the `CHARACTER STRING` type and defined in Figure 14.4 on page 307. The embedded character string is encoded[14] with respect to the transfer syntax identified by the `identification` component, and then transmitted by the `string-value` component of type `OCTET STRING`.

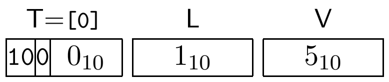
18.2.22 Information objects and object sets, encoding of a value of an open type

Information objects and object sets are never encoded. To transfer the information they contain, it must be extracted as exposed in Section 15.6 on page 336. This information can then be used by other ASN.1 values and encoded according to the rules associated with the type of these values.

To transfer a value of an open type (or of type `ANY` for the specifications referring to ASN.1:1990), a value of a tagged type (see Section 18.2.16 on page 409) is encoded with the tag of the actual type of the value. Thus the value:

```
v ABSTRACT-SYNTAX.&Type ::= [0] IMPLICIT INTEGER:5
```

is encoded using the tag presented before the `INTEGER` type, i.e. `[0]`:



[13] In the BER:1994 standard, the `EMBEDDED PDV` and `CHARACTER STRING` types had several encoding forms called EP-A, EP-B, CH-A and CH-B which were more or less computationally expensive (see footnote 4 on page 303). Unfortunately, under some circumstances, chaining such encodings by relay-applications could come up with non-decodable data (notably in the case of the X.500 Directory associated with a DER global encoding, or when these types appeared in extensions). These different forms were finally replaced by a single form as described above.

[14] See footnote 13.

But this tag `[0]` is not enough for the receiver to decode the TLV triplet as an `INTEGER` value. There should be another mechanism of denoting unambiguously the type: it consists in transmitting, with the value of the open type, an integer or an object identifier (in most cases). Both the encoder and the decoder have access to the same table for associating types to object identifiers (or integers). They only need to look up the identifier in this table to find the type associated with the identifier they have received and decode the value of the open type according to its actual type. This association of transmitted informations is formalized in ASN.1 with a component relation contraint (see production *ComponentRelationConstraint* on page 350).

As explained in Section 15.5 on page 329, ASN.1 has offered since 1994 a concept for formalizing this association table: the information object set. Such a table can now be directly taken into account by compilers to generate decoders that can automatically decode values of open types according to their actual types. Regardless of this table, the decoder transfers a value of an open type as an octet string[15] to the receiver. It is then down to the latter to invoke the appropriate decoding routine.

18.2.23 Value set

A value whose type is defined as a value set (production *ValueSetTypeAssignment* on page 110) is encoded according to the rules associated with the type of the value set.

18.3 Properties of the BER encoding rules

First the BER are architecture-independent: the bit weights have been arbitrarily set (see Figure 18.1(c) on page 395) and the encoding formats can be easily converted to those handled by communicating applications. Besides, the encoding is general enough to support integers whatever their lengths; this is definitively a remarkable advantage when comparing to other transfer syntaxes, such as the XDR for example, discussed in Chapter 24.

[15] In many encryption applications, the values of open types cannot be decoded on the fly: the octet string is transferred to the receiver for authentication. It is only once the authentication is completed that the string is actually decoded.

The transfer syntax of the BER is quite verbose: the T tag field can often be deduced from the abstract syntax and the L length field is sometimes redundant when a SIZE subtype constraint is applied. This verbosity, however, sometimes offers some noteworthy advantages. Since it preserves the structure of the abstract syntax, the BER transfer syntax makes upgrading the protocol easier thereby ensuring the upward compatibility.

The specification can, therefore, be extended replacing a type by a CHOICE where one of the alternatives is this very type since a CHOICE is 'encoded' according to the alternative that is chosen:

```
T ::= IA5String                         -- old version
T ::= CHOICE {
   iA5String       IA5String,
   universalString UniversalString } -- new version
```

Besides, if a decoder generated according to some old version of a specification (the IA5String type) receives a value for the universalString alternative of the CHOICE, it has enough information to discard this value since the length of its encoding V is given by the L field: it can ignore L octets and carry on decoding the other octets of the stream.

For the same reason, a decoder can ignore unexpected extra fields in a SEQUENCE or SET type. In other words, the BER have always implicitly[16] supported the concept of extensibility which was introduced in 1994 with the "..." extension marker. The ENUMERATED types and the subtype constraints can be extended while preserving the encoding compatibility since the transfer syntax of the BER does not depend in any way on the boundaries of an interval nor on size limits.

If the complete abstract syntax is tagged in EXPLICIT mode (see Section 12.1.3 on page 213), the encoding includes tag fields T that contain the tags of class UNIVERSAL of the ASN.1 types. A receiver may then decode the stream without knowing the abstract syntax (i.e. the ASN.1 specification) and display the data in a more user-friendly layout: a boolean value can be presented as 'TRUE' or 'FALSE', the components of a sequence may be neatly printed out one above the other...

The systematic presence of the length field L allows the user (if the compiler supports it) to choose the most appropriate size for communicating applications, should this be absolutely required (for example, an even length for 16-bit architecture computers).

[16]It is, in fact, the Presentation layer that should ignore undefined elements [ISO8823-1, clause 8.5.1.a].

Even after the additions and modifications of the ASN.1 and BER standards in 1994 and 1997, these encoding rules have of course remained compatible with those of 1990. Generally speaking, the migration of a module from ASN.1:1990 to ASN.1:1994/97 (see Section 6.4.2 on page 73) preserves interworking. The fact that the BER have been for quite a long time the only encoding rules associated with ASN.1 favoured the development of numerous compilers, some of which are in the public domain.

In some cases, these advantages may turn into drawbacks: since many values are not encoded on a fixed length, the decoder takes some time to pick them out from the stream; so the encoding and decoding procedures often tie up most of the computing resources to convert the data from the internal data model to the transfer syntax's format and reverse. The running time is obviously a crucial issue for real-time applications and high speed networks. [Lin93] shows that, under some realistic conditions, it is possible to use very fast encoders and decoders for BER. As for the encoding size, we will see in Chapter 20 the notable improvement induced by the packed encoding rules (PER).

The consequence of this variety of options for the implementation is that a decoder must support all these different possibilities whereas an encoder may offer only a few of them. For this reason, decoding is generally computationally more expensive than encoding. To get round this difficulty, one of the specializations of the BER presented in the next chapter can be used.

18.4 A complete example

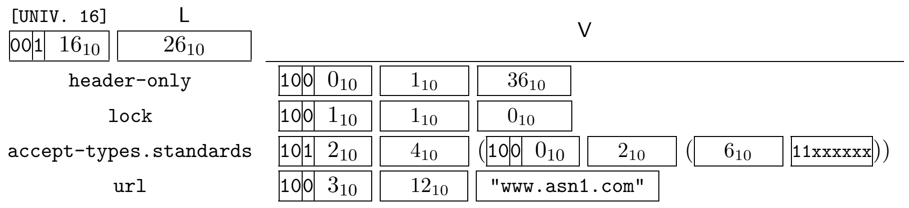
To conclude this chapter, we propose a complete encoding of a relatively simple type to illustrate the aspects of the BER discussed earlier[17]:

```
MyHTTP DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
GetRequest ::= SEQUENCE {
  header-only  BOOLEAN,
  lock         BOOLEAN,
  accept-types AcceptTypes,
  url          Url,
  ... }
```

[17]This example is adapted from http://www.w3.org/Protocols/HTTP-NG/asn1.html.

```
AcceptTypes ::= SET {
  standards BIT STRING {html(0), plain-text(1), gif(2),
                        jpeg(3)} (SIZE (4)) OPTIONAL,
  others    SEQUENCE OF VisibleString (SIZE (4)) OPTIONAL }
Url ::= VisibleString (FROM ("a".."z"|"A".."Z"|"0".."9"|
                             "./-_~%#"))
v GetRequest ::= {
    header-only  TRUE,
    lock         FALSE,
    accept-types { standards {html,plain-text} },
    url          "www.asn1.com" }
END
```

Since all the components of the `GetRequest` and `AcceptTypes` types are automatically tagged (in the context-specific class) by one-increment starting from 0, the value `v` is therefore encoded as the following octet string (quite informally represented):



The BER encoding and decoding of this value, using first the definite-length form and second the indefinite-length form, are simulated in Appendix A on page 499.

Chapter 19

Canonical and Distinguished Encoding Rules (CER and DER)

Contents

Whatever the preference my esteem should be based on,
To estimate all is to estimate none [...]
I want to be distinguished; and to put it bluntly,
To please human kind bears no interest to me.

Molière, *The Misanthrope.*

In 1992, the need for encoding rules that could be rid of the BER encoding options became more and more urgent, particularly for applications that relayed information associated with a digital signature but also because these numerous encoding options made compliance tests for protocols time-consuming and expensive (see Section 23.2 on page 480).

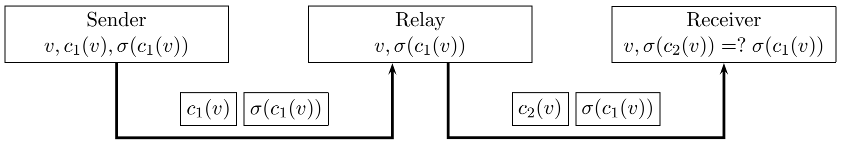


Figure 19.1: Relay of a message v and its digital signature σ with an initial encoding c_1 and second encoding c_2

The purpose of the Canonical Encoding Rules (CER) and the Distinguished Encoding Rules (DER) was to meet those demands by specializing the basic encoding rules (BER); they were included in the [ISO8825-1] standard in 1994.

19.1 A need for more restrictive rules

This need was at first put forward by X.400 e-mail and X.500 directory developpers who wanted to provide a digital signature when transmitting a value to make sure the bit string was not altered during transfer. The X.509 digital signature can[1] be obtained by applying a hash function on the bit string generated by the encoder, and then by encrypting the result with a private key.

Figure 19.1 presents the problem sometimes encountered if the data are relayed by some intermediate systems. If the sender has to transmit the value v, it first encodes it and computes the digital signature σ of the resulting encoding $c_1(v)$. This encoding $c_1(v)$ is then sent along with the digital signature $\sigma(c_1(v))$. The relay decodes the value v and keeps the digital signature[2]. It re-encodes v using a procedure c_2 possibly different from c_1 and associates the digital signature $\sigma(c_1(v))$ with the encoding $c_2(v)$ obtained. The receiver, who knows the encryption key computes the digital signature $\sigma(c_2(v))$ of the bit string and compares it with the digital signature $\sigma(c_1(v))$ of the sender.

[1]Most other signature schemes, such as elliptic curve, do not encrypt the hash. But this point is not important here, for this book is not dedicated to digital signatures and encryption.

[2]We obviously assume that the relay is not part of the secured architecture, so that it does not know the encryption key.

The pre-requisite necessary for the principle of secured data transfer is that the relay re-encodes[3] the value v in the *same* way as it was received, i.e. $c_1(v) = c_2(v)$ for all value v. But as we have seen in the previous chapter the BER encoding rules give way to degrees of freedom when implementing an encoder and when encoding: for example, if the value v in Figure 19.1 on the preceding page includes the boolean `TRUE`, it is impossible to ensure that the BER encoder of the sender and that of the relay will associate the same (non-null) octet with this value. This counter-example is enough to conclude that the BER cannot meet the specific needs of e-mail and directory applications.

Generally speaking, encoding rules that leave no degrees of freedom, whether it is for their implementation or dynamically during the encoding, are called *canonical*. Two sets of canonical rules derived from the BER were standardized in 1994: the Canonical Encoding Rules (CER) and the Distinguished Encoding Rules (DER). These are specializations of the basic encoding rules, which means that a BER decoder can decode a bit stream generated by a CER or DER encoder. The reverse is obviously false.

The two sets of encoding rules offer the interesting property of establishing a bijection between abstract values (of a given PDU) and their encoding: for every ASN.1 abstract value we can map a single octet string and vice versa, for any octet string there exists a single corresponding abstract value. Using this property, the receiving application can compare the octet stream it receives with a given octet string without knowing the value it respectively corresponds to (and even ignoring the abstract syntax of the protocol involved).

The key difference between the two sets of encoding rules is that the CER associate an indefinite-length format with the constructed form encoding whereas the DER use a definite-length format. As a result, the CER are mainly aimed at applications which have to transfer great amount of data. Table 19.1 on the next page summarizes the restrictions imposed by the CER and DER to the BER.

[3]It is interested to notice that, should the relay and the receiver not first decode then reencode what they receive, there would be no reason for using canonical encoding rules such as DER or CER.

CER	**DER**
Length field `L` (see Figure 18.3 on page 396)	
- `L` is in definite length on the minimum number of octets if the encoding is in primitive form (short definite form if `L` $\leqslant$ 127, long definite form otherwise) - `L` is in indefinite length if the encoding is in constructed form	- `L` is in definite length on the minimum number of octets whether the encoding is in primitive or in constructed form
`BOOLEAN` **value**	
- the `TRUE` value is encoded as $\boxed{\texttt{11111111}}$	
`REAL` **value** (see Figure 18.5 on page 401)	
- if the `base` component equals 10, a real value is encoded as a character string of NR3 format without spaces [ISO6093]; it has no "`+`" sign if the number is positive; the dot is the decimal separator; the mantissa must not begin nor end with zeros; it is followed by a dot and the `E` character; the exponent must not use the "`+`" sign nor begin with a `0` but for the null exponent denoted "`+0`"	
	- [X.509] forbids `base-10`-real values in the Directory applications
- if `base` = `2`, a real value is binary-encoded with mantissa `M` and exponent `E` so that the mantissa equals `0` or is an odd number; the mantissa and the exponent are encoded on the minimum number of octets; the scale factor `F` equals `0`	
`BIT STRING` **value**	
- the unused bits in the last octet equal `0` - if the type includes a named bit list, the trailing `0` bits are not encoded; if the type includes a `SIZE` constraint, the value delivered by the decoder to the application must nevertheless respect it (i.e. trailing `0` bits are added if necessary); if the string includes no `1`-bit, it is encoded as a value of length `1` as an octet set to `0`	
`BIT STRING` **or** `OCTET STRING` **value, character string**	
- if the string comprises less than 1,000 octets, the encoding is in primitive form - otherwise it is encoded as a constructed form and every subdivision but the last (if needed) consists of 1,000 octets	- primitive form encoding only
`GeneralString` **value**	
- escape sequences must only be used when the requested character set differs from the usual C0, C1 and G sets (see also the defect report http://www.furniss.co.uk/maint/asn/dr8825_1_005.htm)	

.../...

Table 19.1: CER and DER restrictions on BER

CER	DER
.../...	
`GeneralizedTime` or `UTCTime` value	
- seconds are not compulsary; no meaningless `0` in fractions of seconds; no decimal dot if there is no fraction of seconds - the value must be in universal time coordinate (“`Z`” suffix) - for the `GeneralizedTime` type, the decimal dot “`.`” must be used	
`SEQUENCE` or `SET` value	
- the components that equal their default value are never encoded (all the ASN.1 compilers may not be able to perform this test if the constructed types are too complex)	
`SET` value	
- the components are encoded in the canonical ascending order of their tag (see rule ⟨15⟩ on page 228); if the module includes the clause `AUTOMATIC TAGS` in its header, the order of the specification is kept	
- when a component is an untagged `CHOICE`, it is ordered as if it had the smallest tag of its alternatives (*static* sort)	- when a component is an untagged `CHOICE`, it is ordered as if it had the tag of the alternative retained in the value (*dynamic sort*)
- to avoid this sort step, it is recommended to use a `SEQUENCE` type in the ASN.1 specification instead	
`SET OF` value	
- the elements are sent in the ascending order of their encoding: these encodings are compared as octet strings adding trailing null octets if needed for comparison's sake only - to avoid this *dynamic* sort, it is recommended to use a `SEQUENCE OF` type in the ASN.1 specification instead	
`EXTERNAL`, `EMBEDDED PDV` or `CHARACTER STRING` value	

- the encoding can be relayed only if the abstract syntax includes subtype constraints to avoid transferring the presentation context identifier, for example:
`EMBEDDED PDV (WITH COMPONENTS {...,`
`  identification (WITH COMPONENTS { ...,`
`    presentation-context-id ABSENT,`
`    context-negotiation ABSENT })})`
- character transfer syntaxes associated with character abstract syntaxes registered under `{iso standard 10646 level-1(1)}` are canonical; otherwise, values that conform to abstract syntaxes registered under `level-2(2)` or `level-3(3)` can still be relayed	

Table 19.2: CER and DER restrictions on BER (continued)

19.2 Canonical Encoding Rules (CER)

Historically, the canonical encoding rules were designed after the distinguished encoding rules. They were registered with the object identifier {`joint-iso-itu-t asn(1) ber-derived(2) canonical-encoding(0)`} in the registration tree of Figure 10.4 on page 161 and are documented by the object descriptor `"Canonical encoding of a single ASN.1 type"` (see Section 11.15 on page 198).

They are more particularly suited for applications with potentially important encodings such as the office document architecture (ODA), for example. Indeed, if the encoded value becomes larger than the memory space available for the encoder, it is possible to start emitting before having completed the whole value's encoding since the CER encode the constructed values' length in indefinite format.

The differences between the CER and the BER are summarized in Table 19.1, which starts on page 420. The canonical encoding rules are hardly ever used in practice because few compilers can generate such encodings. This is therefore the DER that are generally considered as the canonical encoding rules by default.

19.3 Distinguished Encoding Rules (DER)

The DER derive from the constraints imposed on the BER by the standard of the Directory authentication framework [X.509, clause 8.7]. They are registered with the object identifier {`joint-iso-itu-t asn1(1) ber-derived(2) distinguished-encoding(1)`} in the registration tree of Figure 10.4 on page 161 and are documented by the object descriptor `"Distinguished encoding of a single ASN.1 type"` (see Section 11.15 on page 198).

They are particularly designed for secured data transfer, and more specifically when a digital signature is used. For this reason, the DER have an up-and-coming career in the context of electronic business on the Internet. They can also be used by applications that need transferring data of average size. Indeed, the application needs to have enough memory space for encoding a value because the DER systematically use the definite length format. This can be an asset for the receiving applications that may need to ignore part of the stream: since each segment is preceded by its length it is easy to know the number of octets to discard.

The differences between the DER and the BER are summarized in Table 19.1, which starts on page 420. The DER encoding/decoding of the example of Section 18.4 on page 415 is simulated in Appendix A on page 499.

Chapter 20

Packed Encoding Rules (PER)

Contents

Monsieur Jourdain. — Mahametta per Jordina.
Madame Jourdain. — What does this mean?

Molière, *Le Bourgeois gentilhomme.*

The criticism expressed towards the Basic Encoding Rules regarding their cost in terms of size (50% extra cost in average compared to the actual data to encode) naturally lead to the development of the Packed Encoding Rules or PER. The gain[1] in size is 40 to 60% at least compared

[1]There exists, however, a few (totally useless) cases, in particular when using extensibility, where the BER encoding prove less expensive than its counterpart in PER.

with the BER encoding of the same protocol (not necessarily optimized for PER encoding).

These encoding rules are therefore particularly appropriate for protocols that need to transfer data at a high rate in domains like telephony over the Internet, videophone and multimedia in general, to mention but a few.

20.1 Main principles of PER

The golden rule of the PER can be stated as follows: 'obtain the most compact encoding using encoding rules as simple as possible'. To do so, a tool for generating PER encoders and decoders applies to the ASN.1 specification more accurately than it does for BER. We shall see that while providing a more compact encoding for ASN.1 primitive types than the BER, a PER compiler also relies on some subtype constraints of the specification to go further in this compression. The impatient reader may refer straight away to Table 20.1 on the next page to be convinced.

Instead of using a systematic recursive format in triplets TLV ⟨tag, length, value⟩ of the BER, the PER format could be interpreted as '[P][L][V]' ⟨optional preamble, optional length, optional value⟩ where the fields P, L and V are no longer series of octets but series of bits. The rest of this chapter will present this format in detail but it is already straightforward that by giving up the systematic use of the TLV format, we remove the BER's criticized overload due to the octets of tag and length fields (T and L). A contrario, the absence of a length field (and even of a value field in some cases) induces a loss of boundaries in the encoding; the receiving application will be able to decode the bit stream only by referring to the ASN.1 specification.

Without the T tag field the PER encoding has no longer the property of implicitly supporting the extensibility; this extensibility should therefore be planned and stated from the very first version of the specification by inserting the extension markers "..." in every type that may be extended in the future (see Section 12.9 on page 244).

	BER definite length	**PER aligned variant**
`v INTEGER (123456789..123456792) ::= 123456790`	6 octets	2 bits
`v INTEGER (123456789..MAX) ::= 123456790`	6 octets	2 octets
`v INTEGER ::= 123456790`	6 octets	5 octets
`v IA5String (SIZE (4)^FROM ("ACGT")) ::= "TGAC"`	6 octets	1 octet
`v IA5String (FROM ("ACGT")) ::= "TGAC"`	6 octets	2 octets
`v IA5String (SIZE (4)) ::= "TGAC"`	6 octets	4 octets
`v IA5String ::= "TGAC"`	6 octets	5 octets
`v SEQUENCE OF BOOLEAN ::= {-- 64 elements --}`	195 octets	9 octets
`v SEQUENCE SIZE (64) OF BOOLEAN ::= {-- 64 elements --}`	195 octets	8 octets
`v SEQUENCE OF INTEGER (0..65535) ::= {-- 64 elements --}`	195 octets	129 octets
`v SEQUENCE { a INTEGER (0..7), b BOOLEAN, c INTEGER (0..3), d SEQUENCE { d1 BOOLEAN, d2 BOOLEAN }} ::= { a 5, b TRUE, c 1, d { d1 TRUE, d2 TRUE }}`	19 octets	1 octet[a]

[a]This example demonstrates that a PER encoder is sometimes more interesting (and remains generic) than a bespoke encoder developed 'by hand' to optimize the transmissions for a particular protocol.

Table 20.1: Comparison of encoding size between BER and PER

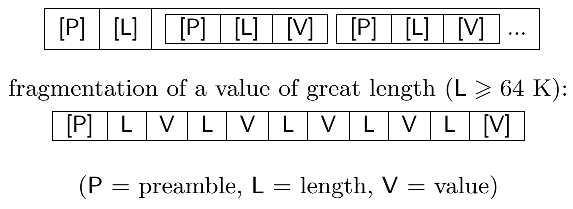

[P] [L] [P] [L] [V] [P] [L] [V] ...
fragmentation of a value of great length (L ⩾ 64 K):
[P] L V L V L V L V L [V]
(P = preamble, L = length, V = value)

Figure 20.1: Recursive formats of the PER transfer syntax

It is now clear that tags are never encoded in PER[2]. A length field `L` is encoded only if the size has not been fixed by a `SIZE` subtype constraint in the ASN.1 specification or if the data size is important (see Figure 20.1). The encoding of values of type `SEQUENCE` or `SET` is preceded by a bit-map which indicates the presence or absence of optional components. Similarly, an index indicates the alternative retained in a `CHOICE` type before encoding the value associated with this alternative.

Though not designed at first for this purpose, the PER provide encoders and decoders that generally take less processing time than their BER counterparts, which goes against a common belief (it obviously depends on the ASN.1 specification, but twice as fast procedures are not unusual). Indeed, many factors are determined statically, i.e. once for all during the compilation stage, and integrated in the encoder and in the decoder. Besides, transfer is also faster than for BER because the lowest layers of the network stack deal with shorter frames. There are few PER compilers on today's market (and none in the public domain) probably because they are much more difficult to develop (and maybe also because the [ISO8825-2] standard is not quite so easy to read!).

20.2 The four variants of encoding

The packed encoding rules break down into two categories: basic and canonical, and either can be of the aligned or unaligned variant. The advantages of *the canonical form* are those already mentioned in Chapter 19 for the canonical and distinctive encoding rules: this form is more specifically adapted to relay systems and secured applications for which

[2]They remain, however, compulsary in the ASN.1 specification (they are used for example to order the components when encoding in PER a value of type `SET`), even though an `AUTOMATIC TAGS` clause in the module header spares the trouble of complying with the condition on distinct tags.

values are authenticated by a digital signature. In basic form, the abstract value may have several PER encodings whereas it has only one in canonical form. On the other hand, for the tests are limited, the basic-form encoder will run faster than a canonical encoder.

In *aligned variant*, padding 0 bits are inserted when needed to restore the octet alignement[3]. The *unaligned variant* is far more compact but encoding and decoding require much more time-consuming processing. In the second variant, the octet alignment is never re-established, which means that all the bits are used without exception. Only the whole protocol data value (PDV) is (often but not necessarily) padded with trailing zero bits so that the overall number of bits could be a multiple of 8 and could, therefore, be stored in a file, transmitted through a network or used as a value for an open type (see Section 20.6.11 on page 445).

If the complete encoding is empty (when the type is restricted to a single value for example) a null octet is transmitted.

The aligned and unaligned variants cannot interwork, that is to say that an aligned-variant decoder (say) cannot decode the bit stream sent by an unaligned-variant encoder. The basic unaligned variant is the most compact. In decreasing order, then we have the canonical unaligned variant, the basic aligned variant and the canonical aligned variant.

In the context of a transfer syntax negotiation on the Presentation layer (see Figure 3.2 on page 22), we indicate the variant of PER encoding to be used with one of the four object identifiers beginning with `{joint-iso-itu-t asn1(1) packed-encoding(3)}` presented in Figure 10.4 on page 161. The PER variants supported by a protocol can be specified in an international standardized profile (or ISP, see footnote 3 on page 378).

20.3 PER-visible subtype constraints

In order to compress at best the encoding, the PER rely on the numerous subtype constraints of the ASN.1 specifications. These allow the encoding procedure to limit or even suppress the length field `L` and the value field `V`. When dealing with protocols for which data transfer rate is an important factor, the specifier should take care of introducing sensible subtype constraints to obtain the most compact encoding as

[3]In Sections 20.4 and 20.5, only the length and value fields mentioned 'octet-aligned in aligned variant' are preceded by padding 0 bits so that they start on an octet boundary.

possible (Table 20.1 on page 427 has nevertheless shown that the gain was already noticeable for a non-optimized specification).

We shall see that the constraints taken into account by PER are those which are the most frequently used in ASN.1 specifications: this choice allows easier implementation of PER compilers (or development of encoding and decoding routines 'by hand'). For the same reason, only the most simple constraint combinations (using set operators) have been retained. Besides, these constraints do not slow down the encoding or decoding real-time process since their evaluation is statically done when compiling.

The only *PER-visible constraints*, i.e. the constraints that PER take into account, are:

- single value and value range constraints (both of which may be composed with the vertical bar "`|`") if applied to an `INTEGER` type:

  ```
  T1 ::= INTEGER (40|50)
  T2 ::= INTEGER (25..30)
  ```

  In case of set combinations of constraints, these are reduced by the compiler to a single constraint called *effective*, which is the smallest interval that satisfies the constraint combination ;

- constraints by type inclusion applied to an `INTEGER` or character string type if the type referenced in the constraint includes only PER-visible constraints:

  ```
  T3 ::= INTEGER (T2)
  C1 ::= IA5String (FROM ("A".."Z"))
  C2 ::= IA5String (C1|FROM ("a".."z"))
  ```

- size constraints:

  ```
  T4 ::= PrintableString (SIZE (100..120))
  ```

  In case of set combinations of size constraints, these are reduced by the compiler to a single *effective* constraint, which is the smallest interval that satisfies the constraint combination[4]; thus:

  ```
  T5 ::= BIT STRING (SIZE (1..4)|SIZE (10..15))
  ```

[4]The notion of effective constraint is not quite defined like this in the [ISO8825-2] standard. Our definition is simpler and amounts to the same thing in terms of encoding. Indeed, it is the interval that matters and not the number of its values that are actually used.

is reduced (only for encoding's sake) in:

```
T5 ::= BIT STRING (SIZE (1..15))
```

(the fact that there can be no strings of type `T5` with a length between 5 and 9 is not significant);

- alphabet constraints applied to a known-multiplier character string type (see Table 11.1 on page 175) if it includes no extension marker:

  ```
  T6 ::= NumericString (FROM ("0".."9"))
  ```

  in case of set combinations of alphabet constraints, these are reduced by the compiler to a single *effective* constraint that encompasses[5] all the contraints:

  ```
  T7 ::= IA5String (FROM ("AT")|FROM ("GC"))
  ```

  is reduced (only for encoding's sake) to:

  ```
  T7 ::= IA5String (FROM ("ATGC"))
  ```

- set combinations (using `UNION`, `INTERSECTION`, `ALL` and `EXCEPT` operators) of PER-visible constraints[6] will be reduced by a compiler to a single *effective* constraint such that the value set it allows could be the smallest interval (for size or single-value constraints) or the smallest set (for alphabet constraints) that includes the result of the combination of constraints; for example, the effective constraint associated with the type:

  ```
  T8 ::= INTEGER (T1|T2)
  ```

[5]This method for computing the effective constraint is not quite that of the standard (more particularly what can be found in [ISO8825-2, Section B.2], though not a normative part of the Standard), but this standard is very likely to be amended according to what is presented here.

[6]As a consequence, a type like:

```
C3 ::= IA5String ("TAGC"|SIZE (1..10))
```

has no effective constraint (because the single-value constraint is not PER-visible when applied to a character string type). Similarly, the type:

```
C4 ::= IA5String (C3 ^ SIZE (5))
```

has no effective constraint because the constraint by type inclusion using `C3` includes a non PER-visible constraint (single-value constraint). This rule makes the effective-constraint combinations easier to compute for an ASN.1 compiler.

is `(25..50)` because it is the smallest interval including all the values of type `T1` and all those of type `T2`. In case of combinations of alphabet and size constraint such as:

```
T9 ::= IA5String (FROM ("AB")^SIZE (1..2)
                 |FROM ("DE")^SIZE (3)
                 |FROM ("ABDF")^SIZE (4..5))
```

the effective constraint[7] is `(FROM ("ABDEF")^SIZE (1..5))`. In practice, the constraint combinations encountered in specifications do not have such a complexity and determining the effective constraint is no problem;

- `WITH COMPONENTS` constraint applied to `CHARACTER STRING` and `EMBEDDED PDV` types if it consists in restricting the `syntaxes` component to a single value or reducing the `identification` component to the `fixed` alternative (see Figure 14.3 on page 305 and Figure 14.4 on page 307);

- if a type is followed by multiple constraints one after the other, the PER-non-visibility of one constraint has no effect on the PER-visibility of the other constraints:

```
T10 ::= IA5String (SIZE (1..2))("AB")(FROM ("A".."F"))
```

  this type has an effective size constraint which is `(SIZE (1..2))` and an effective permitted alphabet constraint which is `(FROM ("A".."F"))`;

- if constraints are extensible (and extended), the effective constraints are computed considering only the extension roots of the constraints[8] (even though the extensions may contain constraints that are not PER-visible).

The ASN.1 working group are now considering simplifying the notions of 'PER-visible constraints' and 'constraints extensible for PER encoding' to make them more straightforward and easier to implement. None of these potential changes will of course affect the encoding of constraints that are generally used in specifications.

[8]Note that the set of values that corresponds to the effective constraint can be bigger than the set of values corresponding to the constraints' extension roots. From the PER viewpoint, a value is encoded with its extensibility bit set to `1` if it does not conform to the extension root of the constraint *as described in the ASN.1 specification*.

Type	**PER-visible constraints**
`BOOLEAN`	none
`NULL`	none
`INTEGER`	single value (see on page 260), value range (see on page 263), type inclusion (see on page 261), set combination (see on page 285), constraint extensibility (see on page 291)
`ENUMERATED`	none
`REAL`	none
`BIT STRING`, `OCTET STRING`	`SIZE` (see on page 266), set combination (see on page 285), constraint extensibility (see on page 291)
`OBJECT IDENTIFIER`	none
`NumericString`, `PrintableString`, `VisibleString`, `ISO646String`, `IA5String`, `UniversalString`, `BMPString`	`FROM` (see on page 268), `SIZE` (see on page 266), type inclusion (see on page 261), set combination (see on page 285), extensibility of the `SIZE` constraint (see on page 291)
not-known-multiplier character string types (see Table 11.1 on page 175)	none
`GeneralizedTime`, `UTCTime`, `ObjectDescriptor`	none
open type (see rule ⟨3⟩ on page 347)	none
`SEQUENCE`, `SET`	none
`SEQUENCE OF`, `SET OF`	`SIZE` (see on page 266), set combination, constraint extensibility (see on page 291)
`CHOICE`	none
`EXTERNAL`	none
`EMBEDDED PDV`, `CHARACTER STRING`	`WITH COMPONENTS` (see on page 277) to restrict the alternative `syntaxes` to a sequence of two fixed object identifiers or force the component `identification` to adopt the alternative `fixed`

Table 20.2: PER-visible constraints

The effective constraints presented above can be easily implemented. These constraints are summarized in Table 20.2 on the page before. Remember that subtype constraints apply to abstract values and need to be interpreted to evaluate how much they impact the transfer syntax: the type `UniversalString (SIZE (4))`, for example, contains 4-character (abstract) strings encoded on $4 \times 4 = 16$ octets.

All the other constraints and constraint combinations do not influence the PER encoding of a value. This does not mean, of course, that they are systematically ignored by a compiler. It may use them to generate encoders that check the conformity of the value to be transmitted by the sending application and decoders that detect values that do not respect these constraints. The *non-PER-visible constraints* are in particular:

- single value constraints on types other than `INTEGER`;
- alphabet constraints that include an extension marker and regular expression constraints;
- constraints by type inclusion on types other than `INTEGER` and known-multiplier character string types;
- constraints on not-known-multiplier character string types (see Table 11.1 on page 175);
- combinations (and in particular set combinations) of constraints if some of them (when appearing in the extension root of the combination) are not individually PER-visible;
- constraints introduced by the keywords `WITH COMPONENT` and `WITH COMPONENTS`. Generally speaking, these are not PER-visible because they can make the compiler task too complicated if several constraints apply on the same components;
- constraints including parameters of the abstract syntax;
- table and component relation constraints because the value set resulting from the information extraction is not necessarily fixed (see Section 15.6 on page 336);
- user-defined constraints (introduced by the keywords `CONSTRAINED BY`) and (obviously!) comments.

From the PER viewpoint, a type is *extensible* if it includes an extension marker (for the `ENUMERATED`, `SEQUENCE`, `SET` and `CHOICE` types) or if it is followed by a PER-visible subtype constraint (other than an alphabet constraint `FROM`) that includes an extension marker (see Table 20.2 on page 433). It is important to note that, conversely, all extensible constraints will not necessarily produce an extensibility bit in PER encoding. We will see that this is no contradiction: if a type followed by an extensible non-PER-visible constraint in an ASN.1 specification is not extensible from the PER standpoint then all the values (in the root or in the extensions) can be encoded because all constraints will be ignored.

If the *effective* constraints are altered between two versions of a specification (this is only possible if the specification is modified without following ASN.1 extensibility rules), the encoding is no longer compatible, which implies that if an application has an old version of a specification of the protocol, it will not interwork with another one that would be based on the new version.

A quite subtle case may occur for a specification that would include an extended constraint in its first version, so that the union of the root and the extension would contain all the values of the parent type as in:

```
PositiveInteger ::= INTEGER (0..4, ..., 5..MAX) -- version 1
```

If we assume that values between 1 and 4 are very often sent, such a type has the advantage of providing a very compact encoding (on 3 bits) of these values without forbidding the other (positive) integers.

20.4 Encodings of a whole number

In this preliminary section, we present four forms of encodings, which will be frequently used in the rest of this chapter. Every one of the four forms amounts to the encoding of a non-negative whole number[9].

The non-negative whole numbers will be involved each time we will have to encode a length field `L`, the size of a bit-map indicating the extensions that are present in a `SEQUENCE` or `SET` value, the index of the alternative retained in a `CHOICE`, or the distance from a value of `INTEGER` type to the minimum boundary of a value range constraint.

[9]The standard and self-explicit term ‘non-negative whole number’ has been preferred to the more concise stock phrase ‘natural number’, which is more common in the Number Theory literature.

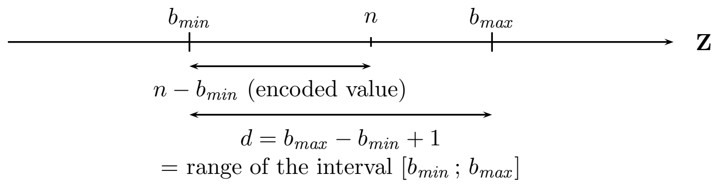

b_{min}
n
b_{max}
Z
$n - b_{min}$ (encoded value)
$d = b_{max} - b_{min} + 1$
= range of the interval $[b_{min}\ ;\ b_{max}]$

Figure 20.2: Encoding principle of a constrained whole number

Let us describe more thoroughly the latter case. We have discussed the PER-visible constraints in the previous section. Every time an ASN.1 specification restricts an `INTEGER` type with an effective constraint that is an interval, say $(b_{min}\,..\,b_{max})$, it proves less costly to encode the difference between the value and the lower bound b_{min}, particularly when this lower bound is large enough (see Figure 20.2).

Constrained whole number encoding

n is a *constrained whole number* if it belongs to an interval whose boundaries b_{min} and b_{max} are finite: $n \in [b_{min}\ ;\ b_{max}]$. Let $d = b_{max} - b_{min} + 1$ be the interval range. If $d = 1$, the only possible value is $n = b_{min} = b_{max}$, it is known by both the sender and the receiver so it is not worth encoding it.

In aligned variant,

- if $2 \leqslant d \leqslant 255$, $n - b_{min}$ is encoded in binary form on the minimum number of bits necessary for representing the range d of the interval, i.e. $\lceil \log_2 d \rceil$ bits (where $\log_2$ represents the logarithm in base 2 and $\lceil\ \rceil$ gives the whole number that equals or is immediately greater than its argument); these bits are appended (without being octet-aligned) to the bit-field to be sent (the length field L is absent);

- if $d = 256$, the offset $n - b_{min}$ is encoded on one octet, which is octet-aligned when appended to the bit-field (the length field L is absent);

- if $257 \leqslant d \leqslant 65{,}536$, $n - b_{min}$ is encoded on two octets, which are octet-aligned when appended to the bit-field (the length field L is absent);

- if $d \geqslant 65{,}537$, $n - b_{min}$ is encoded on the minimal number of octets (which are octet-aligned) necessary for representing the interval range d, i.e. $\lceil \log_{256} d \rceil$ octets; this number of octets is encoded beforehand in a length field `L` as a constrained whole number.

In unaligned encoding, $n - b_{min}$ is encoded in binary form on the minimum number of bits necessary for encoding the interval range, i.e. $\lceil \log_2 d \rceil$ bits; there is no length field `L`.

Semi-constrained whole number encoding

n is a semi-constrained whole number if it equals or is greater than a finite lower bound: $n \in [b_{min}\,; +\infty[$. $n - b_{min}$ is encoded on the minimum number of octets (octet-aligned in aligned variant), i.e. $\lceil \log_{256}(n - b_{min}) \rceil$ octets. This number of octets is first encoded in a length field `L` as described in Section 20.5.

Unconstrained whole number encoding

A whole number is said to be 'unconstrained' if it has no lower bound (even if it is always smaller than an upper bound).

As expected, an unconstrained whole number n is encoded in 2's-complement (and octet-aligned in aligned variant) on the minimum number of octets as explained for the BER encoding rules in Figure 18.4 on page 399. Remember that the principle of 2's-complement representation supports a negative sign for the number n. The number of octets is first encoded in a length field `L` as described in Section 20.5.

Normally small non-negative whole number encoding

This slightly singular terminology denotes the length of the bit-map that ientifies the extensions present in a value of `SEQUENCE` or `SET` type, or the index associated with the alternative retained in a `CHOICE` type. This length is usually quite small but is not limited. It should also be transmitted before encoding, as a value of an open type, the components or alternatives corresponding to extensions in `SEQUENCE`, `SET` or `CHOICE` types.

Let n be a normally small non-negative whole number:

- if $0 \leqslant n \leqslant 63$, a `0`-bit is appended to the bit-field (without being octet-aligned), followed by the binary encoding of n on 6 bits: `0|nnnnnn`[10];

[10] We close these boxes on neither sides to distinguish them from octets.

- if $n \geqslant 64$, a `1`-bit is appended to the bit-field (without being octet-aligned), and n is encoded as a semi-constrained whole number with $b_{min} = 0$ (preceded by its length field `L` as described in Section 20.5): $\boxed{\texttt{1}}\boxed{\ldots\texttt{L}\ldots}\boxed{\ldots n\ldots}$

20.5 Length field encoding

Contrary to the BER for which the length field always represents the number of octets necessary for encoding the value, this field, when present, can be expressed in bits if the value is encoded as a series of bits, in octets if the value is encoded as a series of octets (for the `OCTET STRING` and open types), in characters when encoding a known-multiplier character string or as elements if the value is of `SEQUENCE OF` or `SET OF` type.

Each time the ASN.1 specification limits a type's size with an *effective* constraint of the form `(SIZE (`l_{min}`..`l_{max}`))` (where l_{max} can be $+\infty$), the length l of the value is encoded as a constrained (or semi-constrained if $l_{max} = +\infty$) whole number according to the previous section. In particular, if $l_{min} = l_{max} \leqslant 65{,}535$, the length is not sent since it is known by the decoder. Similarly, if a value is not encoded because it is the only way to proceed in accordance with the specification, the length is null and it is not sent either.

In aligned variant:

- if l is the length of a *bit-map*, $l - 1$ is encoded as a normally small non-negative whole number (see previous section);

- if $l_{max} \leqslant 65{,}535$, l is encoded as a constrained whole number on the interval $[l_{min}\,;\, l_{max}]$;

- if $l_{max} \geqslant 65{,}536$ (i.e 64 K)[11] or if the upper bound l_{max} is infinite (or else if $l \geqslant 65{,}536$):

  - if $l \leqslant 127$, l is encoded on one (octet-aligned) octet whose high-order bit equals `0`: $\boxed{\texttt{0}\,l\,l\,l\,l\,l\,l\,l}$;
  - if $128 \leqslant l \leqslant 16{,}383$, l is encoded on two (octet-aligned) octets whose two high-order bits equal `1` and `0`: $\boxed{\texttt{10}\,l\,l\,l\,l\,l\,l}\ \boxed{l\,l\,l\,l\,l\,l\,l\,l}$, which gives for example if $l = 130$: $\boxed{\texttt{10}\texttt{000000}}\ \boxed{\texttt{10000010}}$;

[11]'K' is the computing constant that equals 1,024.

- if $l \geqslant 16{,}384$ (i.e. 16 K), the encoding is fragmented in units of length $f \times 16$ K, where f equals 1, 2, 3 or 4 (to keep the size of the fragments to a reasonable length), which means that the value is fragmented in packets of 16,384, 32,768, 49,152 or 65,536 units[12]; for each fragment, we determine the greatest value[13] of f such that $f \times 16$ K equals or is smaller than what remains to be sent; before encoding this fragment, we insert a length octet (octet-aligned in aligned variant) for which the two high-order bits are `1` and the six low-order bits represent the value of f. If the last fragment's size is a multiple of 16 K, it is followed by a null octet (which can be interpreted as a length field not followed by a value field), otherwise this last fragment (which is necessarily smaller than 16 K) is encoded with a length value conform to the two previous cases ($l \leqslant 127$ or 16,383). A frame of 147,457 units is therefore fragmented as follows:

`11`	`000100`	65,536 units	`11`	`000100`	65,536 units
`11`	`000001`	16,384 units	`0`	`0000001`	1 unit

In unaligned variant:

- if l is the length of a *bit-map*, $l-1$ is encoded as a normally small whole number;
- if $l_{max} \leqslant 65{,}535$, $l - l_{min}$ is encoded on $\lceil \log_2(l_{max} - l_{min} + 1) \rceil$ bits;
- if[14] $l_{max} - l_{min} \geqslant 65{,}534$ or if the upper bound l_{max} is infinite:
  - if $l \leqslant 127$, l is encoded on 8 bits with the high-order bit set to `0`: $\boxed{\texttt{0}}\boxed{l\,l\,l\,l\,l\,l\,l}$;
  - if $128 \leqslant l \leqslant 16{,}383$, l is encoded on 16 bits where the two high-order bits are respectively set to `1` and `0`: $\boxed{\texttt{10}}\boxed{l\,l\,l\,l\,l\,l}\boxed{l\,l\,l\,l\,l\,l\,l\,l}$;
  - if $l \geqslant 16{,}384$ (i.e. 16 K), the value encoding is fragmented as explained previously for the aligned variant (without being octet-aligned).

[12]The measure unit (bit, octet, character or element) is always known unambiguously by both the sender and the receiver.

[13]Since every fragment is as big as possible, the canonical property is ensured.

[14]This case is slightly different from its equivalent in the aligned variant.

If a type has an extensible size constraint and if the value to be transmitted does not conform to the extension root of this constraint, then the length is encoded as a semi-constrained whole number (with $l_{min} = 0$ and $l_{max} = +\infty$).

20.6 Encoding of all types

20.6.1 `BOOLEAN` value

A `BOOLEAN` value is encoded on a single bit (`1` for `TRUE`, `0` for `FALSE`), which is appended to the bit-field without specifying its length or restoring the octet alignment.

20.6.2 `NULL` value

The `NULL` value is never encoded. If it is the value of an alternative of a `CHOICE` type or of an optional component of a `SEQUENCE` or `SET` type, the bit-map appearing as the preamble of these types' encoding always provides enough information to know whether this value is implied.

20.6.3 `INTEGER` value

Let `(`b_{min}`..`b_{max}`)`, the *effective* range constraint to be associated with the type `INTEGER`. If, in the ASN.1 specification, the `INTEGER` type is followed by at least one PER-visible constraint that is extensible, a preamble consisting of one bit is appended to the bit-field (without being octet-aligned). This bit equals `0` if the transmitted value belongs to the constraint's extension root (*as specified in the ASN.1 module*) and `1` otherwise.

Let n be the integer to send. For the aligned variant as for the unaligned variant, $n - b_{min}$ is encoded as a constrained whole number (or semi-constrained if $b_{max} = +\infty$); if $b_{min} = -\infty$, n is encoded in 2's-complement if it is negative (see Section 20.4 on page 435); if needed (as explained in Section 20.5 on page 438), a length field `L` is inserted, encoded as a constrained whole number on the interval $[1\,;\,l_{max}]$ where l_{max} is the number of bits necessary to encode b_{max}, i.e. $l_{max} = \lceil \log_2 b_{max} \rceil$.

If the value corresponds to an extension of the subtype constraint (*as specified in the ASN.1 module*), the length is encoded as an unconstrained whole number (octet-aligned in aligned variant), then the value is encoded, in 2's-complement if negative (octet-aligned in aligned variant). Thus, the whole number:

```
v INTEGER (3..6, ..., 8..10) ::= 8
```

is encoded as $\boxed{\texttt{1}}\boxed{\texttt{00000001}}\boxed{\texttt{00001000}}$ in unaligned variant.

20.6.4 `ENUMERATED` value

If the `ENUMERATED` type is not extensible, the enumerated whole numbers in the abstract syntax are sorted in ascending order of their values (these numbers may have been computed by the ASN.1 compiler according to the rule ⟨12⟩ on page 140); a new index is then associated with every number, starting from `0` by one-increment. The index of the chosen enumerated number is encoded as a constrained whole number in the interval $[0\,;\text{index}_{max}]$ (see Section 20.4 on page 435). For example, the value:

```
v ENUMERATED {green(0), orange(56), red(2476)} ::= orange
```

is encoded as $\boxed{\texttt{01}}$ in aligned variant as well as in unaligned variant.

If the `ENUMERATED` type is extensible (or if the module includes the `EXTENSIBILITY IMPLIED` clause in its header), a preamble of one bit is appended to the bit-field (without being octet-aligned). If the value appears in the root enumeration, the preamble equals `0`, and the enumerated value is encoded as if the type were not extensible (re-indexing only the root enumeration list). If the value corresponds to one of the extensions, the preamble equals `1`. The enumerated values appearing in the `ENUMERATED` type's extensions are then re-indexed (ignoring the root) by associating a new number starting from 0 by one-increment (the rule ⟨10⟩ on page 140 ensures that every integer associated new extension is greater than the previous one). The extension index that corresponds to the enumerated value to be sent is then encoded as a normally small non-negative whole number.

20.6.5 `REAL` value

The encoding of a real value is the same as for the CER or DER encoding rules (see Figure 18.5 on page 401 and Table 19.1 on page 420). The resulting octets are appended to the bit-field (octet-aligned in aligned variant) once encoded the number of octets in a length field `L`.

20.6.6 `BIT STRING` value

If the type is followed by an extensible PER-visible size constraint, a one-bit preamble is appended to the bit-field (without being octet-aligned). This bit equals `0` if the string length belongs to the constraint's root (*as specified in the ASN.1 module*), and `1` otherwise (in this case, the string length is encoded as a semi-constrained whole number).

Let `(SIZE (`l_{min}`..`l_{max}`))`[15] be the effective constraint associated with the `BIT STRING` type.

- If $l_{min} = l_{max} \leqslant 16$ bits, the string length is not sent and the bit string is appended to the bit-field (without being octet-aligned);
- if $17 \leqslant l_{min} = l_{max} \leqslant 65{,}536$ bits, the string length is not sent and the bit string is appended to the bit-field (it is octet-aligned in aligned variant because it is not necessarily a whole number of octets);
- if $l_{min} = l_{max} \geqslant 65{,}537$ bits, the length field `L` is encoded as a constrained whole number according to Section 20.5, and the value is fragmented as explained on page 439;
- if $l_{min} \neq l_{max}$ (or if there is no effective size constraint), the length field `L` is encoded as a constrained whole number (or semi-constrained whole number when $l_{max} = +\infty$) according to Section 20.5 (if the bit string length is greater than 64 K bits, the value is fragmented as explained on page 439).

If the `BIT STRING` type includes a list of named positions (in curly brackets) in the abstract syntax, all trailing `0` bits are removed; moreover, when the type is followed by a size constraint, trailing `0` bits are added or removed to reach the smallest length which satisfies the size constraint *as it is described in the ASN.1 specification* (this ensures the canonical property for the canonical PER variant).

20.6.7 `OCTET STRING` value

The encoding of an octet string is the same as for a string of type `BIT STRING` but for the unit of length field `L` which is the octet (in particular, if the string is of fixed length greater than 2 octets, its encoding is octet-aligned in aligned variant).

[15] l_{max} may equal $+\infty$.

20.6.8 `OBJECT IDENTIFIER` value

The encoding of an object identifier is the same as in BER (see Section 18.2.8 on page 404). The resulting octets are appended to the bit-field (octet-aligned in aligned variant) once encoded the number of octets in a length field `L`.

20.6.9 `RELATIVE-OID` value

The encoding of a relative object identifier is the same as in BER (see Section 18.2.9 on page 405). The resulting octets are appended to the bit-field (octet-aligned in aligned variant) once encoded the number of octets in a length field `L`.

20.6.10 Character strings and dates

We first describe the case of known-multiplier character string types (see right-hand column of Table 11.1 on page 175). If the type is constrained, it is associated an effective size constraint of the form `(SIZE` (`$l_{min}$`..`$l_{max}$`))[16] and an effective alphabet constraint of the form `(FROM ("`c_1`"|"`c_2`"|`$\cdots\cdots$`|"`c_n`"))`[17]. If the `FROM` constraint is extensible in the ASN.1 module, the effective alphabet constraint gathers together all the characters of the parent type.

If the `SIZE` constraint is extensible in the ASN.1 module, a one-bit preamble is appended to the bit-field (without being octet-aligned). This bit equals `0` if the string length conforms to the root of the size constraint (*as specified in the ASN.1 module*) and `1` otherwise. Remember that if a `FROM` constraint includes an extension marker, it is not PER-visible and does not change the value of this extension bit.

The PER try and compress the encoding of known-multiplier character strings by relying on the *effective* alphabet constraint if present (otherwise the set of all the characters allowed by the type is used). Let n be the number of characters in the effective constraint[18], $b = \lceil \log_2 n \rceil$

[16]See footnote 15 on the preceding page.

[17]Note that contrary to the `SIZE` constraint, what matters here is the number of characters actually used but not the interval that has the smallest lower-bound character and greatest upper-bound character according to the order relation associated with the character string type.

[18]For a string of type `IA5String (FROM ("A"))`, no character is encoded, it is in fact the length field that provides the means of determining the number of characters of the string.

Type	v_{min}	v_{max}	**Encoding standard providing the order relation**
`NumericString`	32	57	[ISO646] (see rule ⟨15⟩ on page 193)
`PrintableString`	32	122	[ISO646] (see rule ⟨17⟩ on page 194)
`VisibleString` `ISO646String`	32	126	[ISO646]
`IA5String`	0	127	[ISO646] (see Table 11.2 on page 178)
`BMPString`	0	$2^{16}-1$	[ISO10646-1] (see rule ⟨47⟩ on page 196)
`UniversalString`	0	$2^{32}-1$	[ISO10646-1] (see rule ⟨46⟩ on page 196)

Table 20.3: Minimal and maximal encoding values for characters of known-multiplier character string types

and $c = \lceil \log_2 b \rceil$. Every character is encoded on $B = 2^c$ bits (the smallest power of 2 immediately greater than b) to preserve the octet alignment in aligned variant and on $B = b$ bits (the smallest number of bits) in unaligned variant. A whole number is associated with every character of the list `("`c_1`"|"`c_2`"|`$\cdots\cdots$`|"`c_n`")` according to Table 20.3. Let v_{min} and v_{max} be the smallest and the greatest of these numbers respectively.

If $v_{max} \leqslant 2^B - 1$, (i.e. if all the characters of the list `("`c_1`"|"`c_2`"|`$\cdots\cdots$`|"`c_n`")` are encoded on B bits), every character of the string is encoded as its associated whole number in the interval `(`v_{min}`..`v_{max}`)`. This avoids re-indexing the characters, which is computationally expensive. Otherwise, the character list `("`c_1`"|"`c_2`"|`$\cdots\cdots$`|"`c_n`")` is re-indexed according to the canonical order defined in the standards [ISO646] or [ISO10646-1] starting from `0` by one-increment. This constitutes the new index, encoded on B bits for every character to be sent. The binary string length is therefore a multiple of B bits in both cases.

If the character string length is fixed ($l_{min} = l_{max}$) and smaller than 64 K, the length field `L` is not inserted and the resulting bit-field is octet-aligned (in the aligned variant) only if $B \times l_{max} \geqslant 17$.

If the character string length is not fixed or if it is greater than or equal to 64 K, the length (in characters) of the string is encoded according to Section 20.5, and appended to the bit-field before encoding the string (it is octet-aligned only if $B \times l_{max} \geqslant 17$ in the aligned variant).

For example, the string:

```
v IA5String (FROM ("ACGT")^SIZE (3)) ::= "TAG"
```

is encoded in $\boxed{\texttt{11}}\ \boxed{\texttt{00}}\ \boxed{\texttt{10}}$. However, the string:

```
v IA5String ::= "TAG"
```

is encoded in the unaligned variant as:

$$\begin{array}{cccc} \texttt{L} & \texttt{"T"}\text{ on 7 bits} & \texttt{"A"} & \texttt{"G"} \\ \boxed{\texttt{0}|\texttt{0000011}} & \boxed{\texttt{1010100}} & \boxed{\texttt{1000001}} & \boxed{\texttt{1000111}} \end{array}$$

and in aligned variant as:

$$\begin{array}{cccc} \texttt{L} & \texttt{"T"}\text{ on }2^3\text{ bits} & \texttt{"A"} & \texttt{"G"} \\ \boxed{\texttt{0}}\boxed{\texttt{0000011}} & \boxed{\texttt{01010100}} & \boxed{\texttt{01000001}} & \boxed{\texttt{01000111}} \end{array}$$

Dates of `GeneralizedTime` and `UTCTime` types (though special cases of the known-multiplier character string type `VisibleString`) are encoded according to BER for the basic PER variant and according to DER for the canonical PER variant (see Table 19.2 on page 421).

Finally, for the character string types that do not belong to the 'known-multiplier' category (including the type `ObjectDescriptor`), no constraint is PER-visible even when one of them is extensible (as a result, the extensibility bit will always be absent). The character string is encoded in BER (see on page 406) in case of basic PER and in DER (see Table 19.1 on page 420) in case of canonical PER. A length field `L` (measured in octets) is inserted beforehand as an unconstrained whole number.

20.6.11 Open type value

An open-type value is actually a value of any ASN.1 type that is known by both the sender and the receiver. This value is therefore encoded according to its effective type with no indication of the type in the encoding of the open-type value. In general, the actual type of the value is referenced by an object identifier that has been sent before. Thanks to an association table (an information object set), the decoder finds the type of the value and can decode the rest of the bit stream according to the corresponding rules. This concept of identification was presented in the chapter on BER on page 412.

The bit-field obtained is completed with padding `0` bits to obtain a whole number of octets (as if it were a PDV). This number of octets is first encoded as an unconstrained whole number in the `L` field and appended to the bit-field (octet-aligned in the aligned variant).

Some of the secure data transfer protocols make a quite sensible use of this systematic length field and specify a type constraint (see production *TypeConstraint* on page 352) of the form:

```
TYPE-IDENTIFIER.&Type (ActualType)
```

Even if, from the abstract syntax viewpoint, the values should be necessarily of type `ActualType`, the presence of the length field makes it possible to draw out the series of octets that correspond to the encoding of the open-type value to hand over the bit-field 'as is' to the receiving application which can apply an algorithm to check the data integrity.

For ASN.1:1990 specifications, a value of type `ANY` is encoded as an open-type value.

20.6.12 `SEQUENCE` value

First, the `COMPONENTS OF` clauses, if present, are developed with respect to rule ⟨19⟩ on page 224. If the `SEQUENCE` type is extensible (or if the module includes the `EXTENSIBILITY IMPLIED` clause in the header), a one-bit preamble is appended to the bit-field (without being octet-aligned). This bit equals `1` if the value includes components that belong to the type's extensions; it equals `0` otherwise.

If the type's extension root (in two parts if two extension markers are present) includes n components marked `OPTIONAL` or `DEFAULT`, a second preamble of n bits is appended (if $n \geqslant 64$ K, this bit-map is fragmented as explained on page 439); it is preceded by its length encoded as a whole number constrained by $l_{min} = l_{max} = n$ as explained in Section 20.5. Every bit equals `1` if the component appears in the value, `0` otherwise (the components are considered in the order they appear in the type definition).

In canonical PER, a component is never encoded if it has the `DEFAULT` value. In basic PER, a constructed-type component (`SEQUENCE`, `SET`, `SEQUENCE OF`, `SET OF`, `CHOICE`) may be encoded (or not) depending on the sender's requirements even if it has the default value; it is never encoded if it is of a basic type[19] (these basic types are those defined in Chapters 10 and 11)

[19]This rule has the advantage of providing a more compact encoding for basic types while avoiding an expensive matching test for more complex types (in practice the default values of complex types are hardly ever used).

After these possible preambles, every component of the extension root is encoded according to the rules associated with its own type in the order they appear in the `SEQUENCE` type definition.

If the type is extensible and if it features p extensions[20], $p - 1$ is encoded as a normally small whole number (each group of components nested in version double square brackets accounts for a single extension). A bit-map[21] of p bits is then added (fragmented if $p \geqslant 64$ K). Each bit equals `1` if the corresponding extension is present in the value (whether this extension is mandatory or marked `OPTIONAL` or `DEFAULT`), `0` otherwise. Then each extension is encoded as an open type value of the appropriate type[22] (see Section 20.6.11).

An extension group in double square brackets is encoded as a `SEQUENCE` type value that would collect all the components of this group (a bit-map is appended in preamble if this group encloses components marked `OPTIONAL` or `DEFAULT`; if all components are absent, the extension is considered 'absent' in the value).

Note that no length field is encoded for a value of type `SEQUENCE`: the bit-map of the optional root components and the bit-map of the extensions enables the decoder to infer the components that are actually present in this value. Of course, a length field may obviously appear in the encoding of some components of the `SEQUENCE` value.

20.6.13 `SET` value

The components of the extension root (in two parts if two extension markers are present) of the `SET` type are sorted according to the canonical order[23] defined by rule ⟨15⟩ on page 228. If a component is an untagged `CHOICE`, it should adopt the tag of its first alternative (in the textual order) for ordering's sake. If the `SET` type includes extensions, their textual order is retained.

[20] If the protocol corresponds to a specific version of the specification, it is recommended to encode the number p of extensions of this version rather than the overall number of extensions the type has in the specification.

[21] This bit-map must not be truncated if it ends with zeros because the sender's protocol may happen to be of an older version.

[22] The extension components must be encoded as open-type values so that a receiver that uses an older version of the ASN.1 specification could ignore this value (or re-send it as an octet string for a relay between two systems): in that case, it knows the number of octets to be discarded.

[23] This canonical order was preferred to the textual order of the components in the specification to prevent interworking problems when operating some 'editorial changes' between two versions of a protocol.

The `SET` type thus ordered is then considered as a `SEQUENCE` type to be encoded according to the rules of the previous section.

20.6.14 `SEQUENCE OF` value

When the size constraint, if present, applied to the `SEQUENCE OF` type is extensible, a one-bit preamble is appended to the bit-field (without being octet-aligned). This bit equals `0` if the number of elements in the list belongs to the root of the constraint (*as specified in the ASN.1 module*) and `1` otherwise. A length field `L` is then encoded to indicate the number of elements of the list if this is not determined by the *effective* constraint ($l_{min} \neq l_{max}$ or the length does not belong to the constraint's root *as specified in the ASN.1 module*).

Finally every element is encoded in order (octet-aligned in aligned variant) according to the rules associated with the element's type.

20.6.15 `SET OF` value

The elements of a set are encoded as those of a `SEQUENCE OF` type without changing the ordering for the basic variant, and once the PER encodings of each element have been sorted in ascending order for the canonical variant (see Table 19.2 on page 421).

20.6.16 `CHOICE` value

If the `CHOICE` type is extensible (or if the module includes the `EXTENSIBILITY IMPLIED` clause in the header), a one bit-preamble is appended to the bit-field (without being octet-aligned). This bit equals `1` if the retained alternative is one of the extensions and `0` otherwise. The root's alternatives are sorted according to the canonical order defined in rule ⟨15⟩ on page 228 (see also note 23 on the preceding page) before associating them with an index starting at `0` by one-increment.

If the `CHOICE` type is not extensible or if it is extensible but it is one of the root's alternatives retained, the index of this alternative is encoded as a value of type `INTEGER` (see Section 20.6.3 on page 440). If the `CHOICE` type includes a single alternative, the index is not encoded. The value of the alternative retained is encoded according to the rules associated with its type.

Another index is associated with each one of the alternatives in the extensions. It is determined by the canonical order of rule ⟨15⟩ on page

228, starting from `0` by one-increment. The index of the alternative is encoded as a normally small whole number after the one-bit preamble, which equals `1`. The version double square brackets do not impact the way this index is computed, nor the value's encoding. The value of the alternative retained is then encoded as an open-type value (see Section 20.6.11).

Contrary to the BER encoding rules, when the PER are used it is impossible to replace a `CHOICE` type in the ASN.1 module by one of its alternatives or to replace a type by a `CHOICE` for which one of the alternatives is this very type.

20.6.17 Tagged type value

As tags are not implicitly supported by the PER, a value of a tagged type is encoded according to the encoding rules that apply to this type. The restrictions on distinct tags must be respected in the ASN.1 module (they are used for instance to order the components of a `SET` type or the alternatives of a `CHOICE` type); it is therefore recommended to insert the clause `AUTOMATIC TAGS` in the module header not to be concerned about these restrictions.

20.6.18 `EXTERNAL` value

For `EXTERNAL` values, the encoding rules of the value field `V` in PER are the same as in BER (see on page 410 and Table 18.2 on page 410). The length of the value field `V` (in octets for the alternatives `single-ASN1-type` and `octet-aligned`, and in bits for the alternative `arbitrary`) is encoded beforehand in a length field `L` according to the rules of Section 20.5 on page 438.

20.6.19 `INSTANCE OF` value

A value of type `INSTANCE OF` is encoded according to the associated `SEQUENCE` type defined by rule ⟨4⟩ on page 358.

20.6.20 `EMBEDDED PDV` or `CHARACTER STRING` values

If the `EMBEDDED PDV` or `CHARACTER STRING` type is followed by a `WITH COMPONENTS` constraint that restricts the alternative `syntaxes` to a sequence of two pre-determined object identifiers (which involves that the abstract and transfer syntaxes are known by both the sender and the receiver) or restricts the component `identification` to the alternative `fixed`[24] (according to the associated `SEQUENCE` type defined on Figure 14.3 on page 305), then the embedded value is encoded in PER as a value of type `OCTET STRING`.

If the `identification` component is not constrained in any of the two ways presented above, the complete value is encoded as if it conformed to the associated `SEQUENCE` type defined on Figure 14.3 on page 305 (in particular, a bit-map of the optional components appears in preamble).

20.6.21 Value set

To encode a value whose type is defined with a value set (production *ValueSetTypeAssignment* on page 110), the type of the set is considered as constrained by the value set itself, according to rule ⟨11⟩ on page 333. A value of the set:

```
Set1 INTEGER (1..20) ::= {1 | 5 | 7}
```

is encoded according to the type:

```
Set1 ::= INTEGER (1..20)(1 | 5 | 7)
```

and consequently with the effective constraint `(1..7)`. In particular, the type associated with the value set:

```
Set2 INTEGER (1..20, ...) ::= {1 | 5 | 7}
```

is not extensible from the PER viewpoint according to rule ⟨7⟩ on page 259.

20.6.22 Information objects and information object sets

Information objects and information object sets are never encoded. To transmit the information they contain, the relevant pieces are extracted as explained in Section 15.6 on page 336. This information is used in other ASN.1 values and encoded according to the rules associated with these values.

[24]In both cases (among others) the complete encoding can be relayed since the value of type `EMBEDDED PDV` or `CHARACTER STRING` (but also `EXTERNAL` type) does not carry the presentation context identifier. The canonical property, however, is obtained only if the embedded value is in turn encoded according to canonical encoding rules [ISO8825-2, clause 7.6].

20.7 A complete example

By way of conclusion for this chapter, we take again the example of Section 18.4 on page 415, so that the BER and PER encodings of the same value can be compared.

The aligned PER encoding of the value `v` is 15 octets long. A vertical bar "`|`" indicates the octets' boundaries and the letter "`x`" highlights the `0` bits inserted to restore the octet alignment (padding-bits).

`0`	extensible type `GetRequest`, no extension
`10`	`TRUE`, `FALSE`
`10`	bit-map for `AcceptTypes`
`110|0xxxxxxx|`	bit string `standards`
`0 0001100|`	length of the `url` component
`01110111|`	`"w"`
`01110111|`	`"w"`
...	encoding of each character on 2^3 bits (the PDV has a whole number of octets)

The unaligned PER encoding of the value `v` has 13 octets (compared to 27 octets for BER):

`0`	extensible type `GetRequest`, no extension
`10`	`TRUE`, `FALSE`
`10`	bit-map for `AcceptTypes`
`110|0`	bit string `standards`
`0000110|0`	length of the `url` component
`1110111|`	`"w"` (on 7 bits)
`1110111`	`"w"`
`1|110111`	`"w"`
`01|01110`	`"."`
`110|0001`	`"a"`
`1110|011`	`"s"`
`11011|10`	`"n"`
`011000|1`	`"1"`
`0101110|`	`"."`
`1100011`	`"c"`
`1|101111`	`"o"`
`11|01101xxx`	`"m"` (and padding to obtain the PDV)

For the type `Url`, the `FROM` constraint restrains the alphabet of the known-multiplier character string type `VisibleString` to 69 characters.

The smallest power of two that equals or is immediately greater than 69 is 2^7 since we have $2^6 < 69 \leqslant 2^7$, so that in unaligned variant, every character of the URL is encoded on 7 bits. The smallest power of two that is immediately greater than 7 is 8 since $2^2 < 7 \leqslant 2^3$, in order that in the aligned variant, each character of the URL is encoded on 8 bits. The code of each character can be found in Table 11.2 on page 178. There is no re-indexing of the characters since all the character codes listed in the `FROM` constraint are smaller than 2^7.

The PER encoding and decoding of this value in the aligned and unaligned variants are simulated in Appendix A on page 499 (may we recommend you to also have a look at the output generated by the `ossPrintPer()` function and reproduced on page 503).

Chapter 21

Other encoding rules

Contents

What would a universal society be if this had no particular country, if it were neither French, nor English, nor German, nor Spanish [...] or even if it were all of these at once? [...] And what would be its language? With this fusion of societies, would a universal idiom emerge or would there be a business dialect for daily use while every nation would keep their own language? Or else would various languages be understood by everyone?

François-René de Chateaubriand.

This short chapter presents some non-standard encoding rules (at an international level at least) but is not meant to be exhaustive. Anyone can define one's own set of encoding rules[1]. Indeed, it is one of ASN.1 advantages compared to other abstract syntax notations: once written the abstract syntax, the encoding may be changed without adapting the abstract syntax (and all the more so when using an ASN.1 compiler since the appropriate option needs only to be given on the command line).

However, a plethora of transfer syntaxes could lead to new interworking difficulties that ASN.1 lays claim to solve. Indeed, if several encoding rules are actually standardized (or designed by some community of users), ASN.1 compiler vendors will only implement a few of them (if not only the BER as it is often the case today) for development cost reasons.

The ideal is therefore to standardize only a few encoding rules, every one of which have their own specific properties concerning data compression, encoding speed, canonical properties... If a user has a compiler that supports those different sets, the adequate transfer syntax should be determined in accordance with the demands of the protocol it applies to.

21.1 Light Weight Encoding Rules (LWER)

Studies on Light Weight Encoding Rules (LWER) started in Germany around 1985 and went on in France from 1988 on at INRIA (called 'Flat Tree Light Weight Syntax' or FTLWS at the time) [Hui90] [DHV92][2].

The purpose was to offer an alternative to the BER (the only encoding rules available at the time) to transfer data in a faster and more efficient way between two machines of similar architecture. The LWER were meant to support 8, 16 or 32-bit memory-words as well as 'little Endian' or 'big Endian' architectures (see Figure 2.1 on page 9) since permuting the octet ordering when encoding or decoding proves costly in terms of processing time. The LWER had therefore at least six different variants but had to be standardized to prevent an increase in the number of these variants required to meet the specific demands of a manufacturer that wanted to adapt it to their own architectures.

Although the LWER optimized the processing time for encoding and decoding routines, the data compression was no real issue (for the 32-bit

[1]Should the negotiation mechanism presented in Figure 3.2 on page 22 require it, an object identifier is associated with the set of encoding rules according to the procedure described in [ISO8823-1, annex B] (see also Section 10.8 on page 153).

[2]The results of INRIA also lead to the High Speed Coding Rules, HSCR [BS92].

alignment in particular, the number of octets generated by the LWER was greater than the BER). For this, all useless information was removed (contrary to the BER); the tags were encoded only to denote one of the alternatives of a `CHOICE` type, the lengths did appear but only before strings (bit strings, octet strings and character strings) or lists (`SEQUENCE OF`, `SET OF`) and were systematically encoded in definite form (see Figure 18.3 on page 396). All the values of fixed length were encoded at the beginning of the bit-field and a pointer (actually the number of octets to be shifted from) indicated the starting-point of every encoding of variable length.

The encoding routines were 1.6 to 5.8 times faster than the BER [DHV92], but the gain depended as much on the implementation techniques allowed by the LWER than on optimizations operated within the ASN.1 specifications (subtype constraints for example).

Besides, in 1985 it became clear that PER encoding and decoding routines (see Chapter 20) were much faster than expected (the PER were not originally designed for this purpose). That was the *coup de grâce* for the LWER, which were eventually given up in 1997. One of the possibilities for obtaining a significant gain compared to the PER would have been to transmit a memory core dump but this would not have been canonical (remember the canonical property is necessary to compute a digital signature, see Chapter 19), nor secured (the risk of transferring a protected area of memory should be considered).

21.2 BACnet encoding rules

The BACnet encoding rules[3] (Building Automation and Control networks, [Ash95]) were designed by the American Society of Heating, Refrigerating and Air-conditioning Engineers (ASHRAE[4]), an international organization of 50,000 people, and the National Electrical Manufacturers Association (NEMA). They are used to transfer on the Internet data collected from control devices for monitoring of central heating, ventilation, air-conditioning or smoke detectors for buildings located in different places.

The description of a transfer syntax that is relevant to the BACnet encoding rules will benefit from the 'encoding control', which is being

[3] http://www.bacnet.org, ftp://ftp.bacnet.org/Encoding.doc

[4] http://www.ashrae.org/

defined by the ASN.1 working group (see Section 21.6). This would make it possible to use a generic tool for generating the encoding and decoding procedures.

21.3 Octet Encoding Rules (OER)

In 1995, when working on National Transportation Communications for Intelligent Transportation Systems Protocol (NTCIP) and Simple Transportation Management Protocol (STMP), the National Electrical Manufacturers Association (NEMA) learned about ASN.1 and its associated encoding rules, the BER. Ignoring the existence of the PER, they had in mind to develop a protocol that would need a small bandwidth so they developed rules specific to the STMP protocol called NEMA PER. These NEMA PER, however, could do with a few refinements since some mistakes still remain (some encoded values cannot be decoded, all the types of the ASN.1 standard are not supported...).

Adapted from the NEMA PER, the Octet Encoding Rules (OER[5]) are meant to be less expensive than the BER, and easier to understand and to implement, than the standardized PER. The general form of the transfer syntax is a triplet [T][L][V] ⟨optional tag field, optional length field, optional value field⟩, which is indeed half-way compromise between the BER TLV and the PER [P][L][V] formats.

The OER use the abstract syntax to generate a quite efficient encoding but support only the most simple subtype constraints, i.e. the (non-extensible) `SIZE` constraint applied on the `BIT STRING`, `OCTET STRING` and known-multiplier character string types and the (non extensible) value range constraint for the `INTEGER` type, which enables the encoder to send an integer on 1, 2 or 4 octets without indicating the length field L. The tag field T is very often reduced to a single octet because the tags are encoded as if they were in explicit mode (except for the `CHOICE` type). There is no canonical ordering for a `SET` or `CHOICE` type, which reduces the computing time necessary for encoding.

The OER are not expected to be standardized because they are not mature enough and not prototyped (the current document still contains mistakes). We can imagine, however, that a new 'fully-aligned' variant of the PER (see also encoding control exposed in Section 21.6 on page 459) or a new clause `ALIGNED` (which would appear just before a type, like the

[5]http://www.viggen.com/ntcip/documents/oer.rtf

EXPLICIT or IMPLICIT keyword) could take into account some principles introduced by the OER.

It seems, however, that by slightly changing the ASN.1 specifications of the NTCIP protocol, the encoding would conform to the standardized PER without affecting the applications already in use; the new users may then take advantage of commercial ASN.1 compilers already available. Besides, the ISO TC 204[6] technical committee, in charge of the NTCIP protocol in particular, announced that the use of PER would be encouraged for the protocols under their responsibility.

21.4 Signalling specific Encoding Rules (SER)

The Signalling specific Encoding Rules (SER) were jointly introduced by France Télécom R&D, and Nokia [Cha92] [Cha97b]. Their main purpose is to generate automatically, using an appropriate compiler, encoding and decoding functions from an ASN.1 description, for protocols that were not originally written in this notation. The protocols in question are mainly found in the signalling domain: the signalling system No. 7 of the Integrated Services Digital Network (ISDN), GSM access protocols...

The advantage is straightforward: the protocol designer (who is more often used to the low layers of the OSI model) now has, without going to great expenses, a high-level specification language and may use the tools that go along with it (and test-tools, in particular, since protocol testing is generally a long and tedious process for telecommunication operators, see Section 23.2 on page 480).

The idea is to define (by reverse-engineering) the messages and parameters of the protocols involved so that the binary encoding expected by the original specification can be found by applying the SER on the associated ASN.1 specification. These rules can support the majority of the signalling protocols and in particular those where the encoding is based on a simplified version of the TLV triplet of the BER, and may use pointers similar to those of LWER exposed in Section 21.1.

The SER encoding of a value is therefore the sequence of octets obtained when applying transformation rules (to support information generally ignored by the BER: size constraint, systematic transmission of components marked DEFAULT...) and restricting the BER encoding options (primitive form and short definite length whenever it is possible...).

[6]http://www.iso.ch/memf/TC204.html

The Minimum Bit Encoding Rules (MBER) were proposed in the middle of the 80s and had the same goal: to provide an ASN.1 specification which, when encoded, produces the same (minimal) series of bits as those described in the reference document that describes a protocol. The MBER were never standardized but were considered while conceiving the SER and also for the first studies on the PER.

21.5 XML Encoding Rules (XER)

A set of encoding rules, called XER, which translate into XML[7] (eXtensible Markup Language) the values described in ASN.1 is being designed on a dedicated electronic mailing list[8]. On Eliot Christian's initiative of the U.S. Geological Survey, who is working on a global information system, these rules are a priori dedicated to WAIS information database (protocol Z39.50, see on page 87) but could be involved in a broader usage.

The idea is to delimit the ASN.1 values with XML markups of the form <MARK>...</MARK>, which means that a value of type:

```
PDU ::= SEQUENCE { component1 SEQUENCE OF T,
                   component2 U }
```

could be encoded as:

```
<COMPONENT1>...</COMPONENT1>
<COMPONENT1>...</COMPONENT1>
<COMPONENT1>...</COMPONENT1>
<COMPONENT2>...</COMPONENT2>
<COMPONENT2>...</COMPONENT2>
```

These new rules were examined by the ASN.1 working group during their June 1999 meeting in Geneva. It was decided that the group would follow their development. But the new concept of XML Schemas [W3C00], it it gets eventually officialized by the World Wide Web Consortium would provide a means[9] of translating ASN.1 into XML and reverse that would be more powerful than the current low-level typing mechanism provided by DTD (Document Type Definition).

[7] http://www.w3.org/XML/

[8] http://asf.gils.net/xer/

[9] This means is still not as powerful as ASN.1 because it will remain difficult to translate into XML semantic links modeled by ASN.1 information object class which are frequently used in important protocols like the X.500 directory for instance.

The main interest of the ASN.1 working group on this point is to provide a user-friendly and inexpensive way of visualizing ASN.1 documents with a web browser and investigate the opportunity of encoding XML pages in PER to provide a more compact encoding than the default encoding used on the web.

It should be noted that the Wireless Application Protocol (WAP) forum developed a compact binary representation of XML called Binary XML[10] intended to reduce the transmission size of XML documents for more effective uses of such data on narrowband communication channels. One may wonder if it would have not been better to write a special ASN.1 module for handling XML documents and then encode them in PER to benefit from their numerous advantages instead of creating a new set of encoding rules which is less compact than the PER.

21.6 Encoding control

Presented during the January 1999 ASN.1 meeting in Lannion, the encoding control notation (ECN) will allow specifiers to define their own encoding rules by referencing standardized encoding rules and modifying some of their characteristics (for example, one may want to use the whole standardized PER set but for the boolean values that would remain encoded on octets), or even to set up completely new ones.

Encoding control [Wil99] will prove useful in application domains that require a particularly optimized transfer syntax (in terms of size or speed of encoding/decoding): the ISDN User Part of CCITT Signalling System No. 7 (ISUP) or Signalling Connection Control Part (SCCP) recommendations, but also some standards for intelligent transportation systems management or radio interface (the 3GPP[11] project of third-generation mobile phones based on the UMTS standard for example). All these standards describe data transfer as bit- or octet-fields in tables or (English) texts without providing an ASN.1 abstract syntax (except for UMTS that is based on big ASN.1 specifications).

A more systematic use of ASN.1 in the context of these protocols will make them more likely to be used in generic test-tools like those based on TTCN (Tree and Tabular Combined Notation) for example, and will prevent a plethora of informal encoding rules (unfit for validation),

[10]http://www1.wapforum.org/tech/terms.asp?doc=PROP-WBXML-19990815.pdf
[11]Third Generation Partnership Project, http://www.3gpp.org/

generally non-standardized, which may become the exclusive property of a single tool vendor.

ASN.1 can describe abstract syntaxes, but for the time being there exists no formal notation that could define encoding rules[12]. The encoding control will probably be modelled by a new category of modules called `ENCODING-CONTROL`; it will contain the information about the alignment, the padding bits, the computation of length field etc. that defines the encoding to be associated with (some of) the generic ASN.1 standard types or specific types imported from another ASN.1 module.

In addition, a linkage module, called `LINK-DEFINITIONS`, which is in principle very much similar to a `makefile` for Unix systems, will associate one or several modules with one or several encoding control modules (or with standardized encoding rules, like the PER aligned variant for example, together with an encoding control module in which some of these standardized rules are modified).

The encoding control standard (which may be called X.692) should be approved in March 2001. All the material defined in this section is currently being defined by the ASN.1 working group. Some of the characteristics described above are therefore liable to change until the final document is approved.

[12]Some notations that describe transfer syntaxes, like CSN.1[13], do exist but these can only define the concepts related to encoding without going further into the detail of the series of the bits that is represented by this new notation.

Part IV

ASN.1 Applications

Chapter 22

Tools

Contents

And the woodcutters who lost their tools,
Crying out loud to have them back.

Jean de La Fontaine, *Fables*.

Without the appropriate software tools, there would not be much point in using a formal notation like ASN.1. We now come to some of the tools associated with ASN.1. When protocol implementation is being considered, the tool par excellence is the compiler. From a set of modules, it generates automatically the encoding and decoding procedures for the various types defined in the specification. This chapter does not focus on any ASN.1 compiler in particular.

22.1 What is an ASN.1 compiler?

Generally speaking, a compiler is a computing tool that reads a program written in a first language, called 'source language', to translate it into a second language called 'target language' (this language is closer to

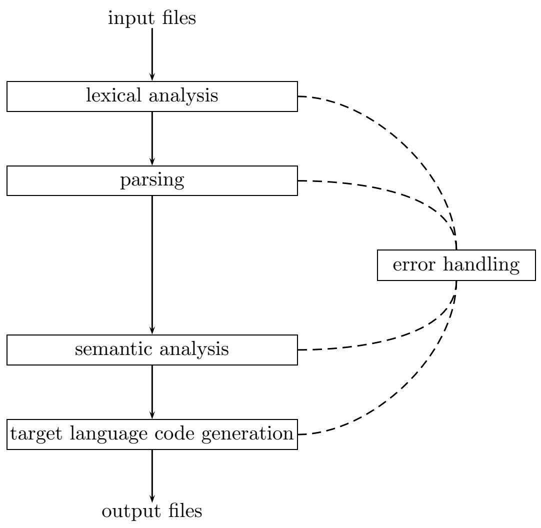


Figure 22.1: Four usual stages of a compilation process

the machine architecture; it is often assembler or some machine-oriented language). In our case, the source language is ASN.1, the target language is generally *C*, *C*++ or *Java*, and the 'program' is a specification made of several modules linked by `IMPORTS` clauses. In this respect, an ASN.1 compiler would better be considered as a *stub compiler*.

The progress of a communicating application designer, who goes from the ASN.1 specification to the executable file that can send and receive data is shown in Figure 22.2 on the next page. We have already breached the subject in Section 4.7 on page 40.

We first have to collect all the files that constitute the ASN.1 specification, including those (transitively) referenced in the `IMPORTS` clauses. Compiler vendors very often have the most common standards and may help you to carry out this task as long as they do not infringe the copyright legislation that may apply to these standards. All these files (whose extension is generally "`.asn`" or "`.asn1`"), each of which contain one or several ASN.1 modules, are given to the compiler as input files.

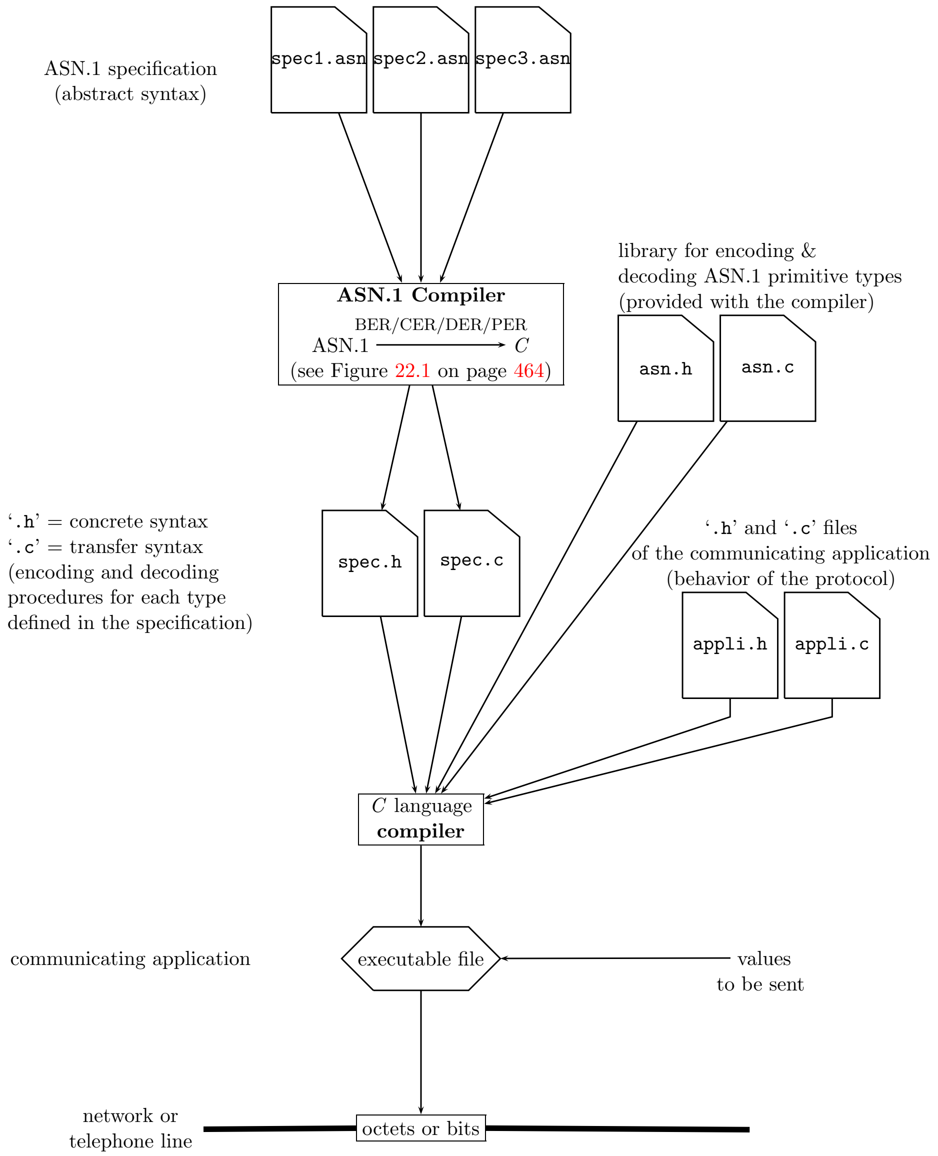


Figure 22.2: Modus operandi of an ASN.1 (to *C*) compiler

In an ideal world[1] of Figure 22.1 on page 464, a compiler breaks down into four layers [ASU86] and each layer can start only if the previous one reported no error: in particular, the semantic analysis is executed only if the files include no lexical or syntactic errors (also called grammatical errors), and the output files are generated if there is no semantic error in the specification.

Parsing and lexical errors are induced by symbols that are not allowed in the ASN.1 grammar (e.g. the underscore "_") or by grammatical structures that are not permitted by the structural rules of the notation (e.g. a comma before a closing curly bracket) while semantic errors denote incoherence in the specification (e.g. allocating an integer to a value declared as `BOOLEAN`).

If the specification includes no syntactic errors and if it is semantically correct, the compiler usually generates:

- a file with the concrete syntax, which is the translation of the data types defined in the ASN.1 specification into the target language (for the most common, the *C* language, this file, called 'header file', has the extension '`.h`');
- one or several files including one encoding procedure and one decoding procedure for each type of the ASN.1 specification; these implement the encoding rules retained (BER, CER, DER, PER...) into the target language and generate the transfer syntax (for the *C* language, these files have the extension '`.c`').

Without further effort, the designers of a communicating application have data transfer procedures at their disposal. What remains to be done is to program the complete behavior of the protocol (for example, tests or actions such as "if the application receives data of type `T`, then it should return an answer of type `U`") or generate it automatically from an SDL specification as we will see in Section 23.1 on page 476.

The files generated by the ASN.1 compiler and those specific to the communicating application are then given to a compiler of the computing language used for programming the communicating application

[1]This ideal should normally be a common feature of all ASN.1 compilers. The outputs of a compiler that complies this decomposition into four distinct layers are more easily read and properly analyzed by the user, who can correct the specification more quickly. Indeed, syntactic errors frequently induce semantic errors and compilers that carry out a semantic analysis of a syntactically incorrect specification tend to come up with useless error messages.

(generally a *C* compiler). This produces an executable file suitable for the machine architecture (which means that it manages correctly the memory alignment, the bit weights...) using libraries provided with the ASN.1 compiler, which contain the encoding and decoding procedures of all ASN.1 primitive types. This executable can send and receive a binary stream on a telephone line or a computer network.

22.2 Notes on compiler usage

The semantic model of ASN.1 standardized in June 1999 (see Section 9.4 on page 121) will prove useful especially for designers of compilers or any other tools since it precisely defines whether two types are compatible, i.e. if a value of one of these types may be used in an expression governed by another.

The encoding and decoding procedures generated by the ASN.1 compiler depend on the tool for a great deal and there would be no point in merely reproducing here an extract for one of them in particular. The interested reader may refer to the URL http://www.oss.com/products/application.html to consult a simple but self-contained example of data encoding and decoding.

The designer may locally refine or guide the compiler's behavior by means of compiling *directives* (to force some length fields to be encoded in defined format when BER are used, to allocate statically or dynamically a list in memory, etc.). In fact these directives are comments (sometimes called 'formal comments') beginning with a specific series of symbols depending on the tool, like "`--*`", "`--$`" or "`--<    >--`".

Several compilers from ASN.1 to *C*/*C*++ (and many other computing languages) can be found on the market and they will not be detailed here. Nevertheless when buying a compiler, it seems important that the potential user consider the following points:

- what are the edition(s) of the ASN.1 standards (1990, 1994, 1997) supported? Does it (really) cover the whole standard(s) in question? Does the compiler impose any restrictions on the standard syntax? (Inserting a semicolon after every definition for example.)

- Is the compiler reliable for checking the semantic consistency of a specification? (A semantic error may induce interworking problems between applications and even the standardized ASN.1 modules are unfortunately far from perfect in this department.) Is this semantic analysis exhaustive?

- What are the encoding rules (BER, CER, DER, PER) furnished by the libraries? Do the BER decoders support all encoding options? (Different length formats, primitive or constructed variant...) Should the project require it, is it easy to switch to another set of encoding rules while keeping the same interface from the communicating-application viewpoint? (Same parameters for invoking the generated procedures, same returned values...)

- Should the project require it, are the encoders and decoders optimized in terms of runnning time? in terms of memory-space?

- Can we insert encoding directives by means of special comments in the ASN.1 modules to restrict the degrees of freedom offered by the BER or change the memory allocation routines? (Encode a particular type in definite form length, store an integer on 4 octets, represent the strings by arrays...)

- Can the compiler automatically generate compliance test procedures to check if the data sent or received respect the subtype constraints of the ASN.1 specification? (Even if the BER encoding does not use these constraints, it may spare the designer considerable time not to have to code the tests by hand if these appear in the formal specification)

- Is a maintenance and technical support available on a real-time basis?

- Will the interface that gives access to the encoding and decoding procedures be kept in the new versions of the compiler? Does the compiler generate an interface that conforms to that of the TeleManagement Forum and X/Open consortium [TMF96]?

Beside the usual *C* or *C*++ code generation, other (sometimes odd) languages may be encountered such as *Pascal*, *COBOL*, *Chill*[2] (CCITT HIgh Level Language)... But for many designers, the most fashionable language, after *C* and *C*++, is now *Java*[3]. The most popular of its applications are X.509 authentication and the H.225 and MHEG standards for multimedia data transfer (see Section 7.4 on page 84).

[2]http://www.kvatro.no/telecom/chipsy/

[3]See, for example, OSS Nokalva's BER, DER and PER Java compiler at http://www.oss.com/products/products.html. Some BER libraries (unfortunately for ASN.1:1990) are also available in the public domain (see http://asn1.elibel.tm.fr/en/links/#java).

22.3 Parsing ASN.1: a troublesome problem

Before introducing some original tools, we need to discuss the common front-end part of all these tools, which constitutes a real programming challenge in the case of ASN.1: parsing (see Figure 22.1 on page 464). A lexical and syntactic analysis takes an input file (which includes ASN.1 modules) as a character stream and generates an abstract syntax tree, which is a memory-resident structured representation of the modules. The execution of the parser points out all the structural mistakes that can be found in the specification. If a specification is syntactically correct, it is represented as a tree-like structure on which a compiler can apply several data processings (a semantic analysis, in particular).

Unfortunately, the grammar of the ASN.1 notation has a structure (use of curly brackets for different concepts, no semicolon at the end of a definition...) which inherently proves quite complicated to deal with when it comes to programming ASN.1 parsers. It can hardly be blamed for it: remember the notation was originally meant to be a means of communication between a standardization committee and application designers. These days are far gone now since encoders and decoders are directly derived from ASN.1 specifications.

If the raw grammar[4] of the ASN.1:1997 standard is analyzed with *Yacc* (Yet Another Compiler Compiler), the most famous bottom-to-top parser generator (called LALR(1)), it issues 396 shift/reduce conflicts and 1,304 reduce/reduce conflicts. *ANTLR* (ANother Tool for Language Recognition)[5], a top-to-bottom parser generator (called LL(k)), indicates more than 200 grammar productions beginning with the same lexical token (an opening curly bracket for example!).

The only way to obtain a good ASN.1 parser is therefore to carry out a long and tedious transformation[6] of the standard grammar to obtain an equivalent grammar, which generates the same language but would have interesting properties for parsing purposes.

[4]Though formally specified in BNF, the ASN.1 grammar as it is described in the standard is not quite appropriate for computing tools. Its designers were rather concerned with making the semantics of the notation's constructions straightforward thanks to self-explicit rule labels.

[5]http://www.jguru.com/thetick/antlrtut/

[6]Note we do not call it conversion: this transformation should be done 'by hand' since the existence of a grammar, with the adequate properties, which would be equivalent to another grammar is an undecidable problem, i.e. there exists no algorithm to carry out the transformation [ASU86].

This transformation into LL(1)-compliant form[7] of the grammars of ASN.1:1990 and ASN.1:1997 are respectively detailed in [Rin95] and [FDD96]. These studies have been the preliminary works necessary for implementing the *Asnp* parser available on the web site associated with this book[8].

The LL(1) grammars have some interesting properties: any syntactic error is guaranted to be flagged as soon as possible during scanning, they enable the compiler to produce more explicit messages and improve the error recovery; they make panic-mode error recovery very easy to implement by removing all the unexpected lexical tokens until a synchronization lexical token is found (closing bracket, closing curly bracket, closing square bracket...); they are easily maintained in the sense that they can integrate some new grammar productions introduced by amendments on the standard for example (this property is mainly due to the fact that the behavior of the parser is directly deduced from the various analysis routines). The main problem of this LL(1) transformation is that it produces a much larger grammar, in terms of rules (with numerous repetitions) than the initial standard grammar.

22.4 Other tools

It is only common sense that a specifier should have an ASN.1 compiler, had it been only to check the syntax and the semantic coherence of the specifications (even more so if they are meant to be included in standards or published). But other tools can help to write specifications of better quality, and in particular:

- a computer-aided syntactic editor, like the Emacs mode[9] available on the web site associated with this book, or a dedicated model for Microsoft *Word*® like the one developed by France Télécom R&D[10], have the following advantages: fast writing (automatic insertion of pieces of ASN.1 code), completion to avoid typing the complete tokens, reduction of spelling mistakes (automatic insertion of some keywords), better readability (highlighting of

[7]Left to right scanning of the input constructing the Leftmost derivation with 1 token of look-ahead.

[8]http://asn1.elibel.tm.fr/en/tools/asnp/

[9]http://asn1.elibel.tm.fr/en/tools/emacs/

[10]If you are interested, please email to asn1@rd.francetelecom.fr.

keywords, bold facing for printouts) and automatic indentation (detection of structural mistakes, straightforward and 'standard' layout) [LD97];

- a pretty-printer for typesetting ASN.1 modules in a homogeneous way according to specific rules: indentation, boldfacing or highlighting of keywords, multiple output formats (text, HTML, LaTeX...) [LD97][11].

These tools should obviously provide numerous configuration options to comply with every user's needs.

France Télécom R&D, developed for their own needs, a specification comparator[12] for ASN.1 [HD98]. Highly parametrable, this tool indicates all the syntactic differences (source of interworking problems) between two specifications. Hence, it can point out the slightest differences between specifications originating from comparable application domains such as fixed and mobile intelligent networks. Using Emacs, one can display side by side the two modules in two different windows where the syntactic differences are highlighted.

The TeleManagement Forum[13] (formerly Network Management Forum[14]) and The Open Group Ltd.[15] (formerly X/Open), a trade association of computer manufacturers that promote the development of portable applications and open system implementations on Unix platforms, specified an ASN.1/*C*++ standard interface based on the object paradigm [TMF96]. It is a group of classes and methods, independent from this tool used for its implementation, which establishes a mapping between ASN.1 types and *C*++ classes. This interface therefore constitutes some link between the abstract syntax on one hand and the concrete syntax on the other. It also makes it possible to define ASN.1 values using *C*++ syntax (real numbers may for example be defined as floats by their decimal representation).

It is highly portable, easy to implement, independent from the encoding rules (BER, PER...) and self-consistent (one routine carries out

[11]See footnote 10 on the preceding page.

[12]See footnote 10 on the preceding page.

[13]http://www.tmforum.org

[14]Although defined in the context of network management (see Section 23.3 on page 482), this interface is independent from the application domain.

[15]http://www.opengroup.org

the same task whatever the ASN.1 type involved). By exchanging *C++* objects with the communicating application, this interface can make the low-level implementation of the encoding rules transparent for the higher levels. These can in fact be chosen dynamically when the application is running together with the associated options (e.g. aligned/unaligned variant for the PER). Finally it is possible to check that the value received or about to be sent conforms to the subtype constraints of the specification.

Interfaces with databases have also been developed: [HSO94] translates an ASN.1 module in a relational database scheme in order to store and search, using queries of the *SQL* language, DNA sequences in the Medline database of NCBI (see on page 92). [HTN] focused more specifically on object-oriented databases. The ASN.1 modules are translated into *C++* classes, which makes it possible to implement an ASN.1 database while limiting joins for information extracting.

The encoding and decoding procedures automatically generated from an ASN.1 specification are generally slower than those implemented directly by a programmer. [Hos96] and [Hos97] propose to include the heuristics used by a programmer when optimizing its code: the basic idea is to predict the frequency with which every type is used relying on a static analysis of the stream and the profile of the types used by communicating applications. The optimization stage strikes a balance between the generated code's size and its running time.

Around 90% of the routine calls are discarded by analyzing only 50% of the generated code. Moreover, the code's size can be cut down by 30 to 50% [Hos93a] using the subtype constraints of the specification (or introducing appropriate constraints) which indicate that an optional component is always (or never) present in the values (`WITH COMPONENTS` constraint) or limit the set of possible values for a type (`INCLUDES` constraint).

Another possible optimization is suggested in [Bla96]: when a decoder receives a message as a bit stream, it compares it with a set of formats generated by all the previous messages. If no correspondence can be found, the message is entirely decoded and a new format is stored in memory; otherwise, an optimized procedure (in which useless tests and dead-end branching have been removed) is used to decode the message more quickly. Tests showed that such an implementation could be 70 to 100 times faster than the unoptimized decoders.

Finally, [WBS90] and [BS93] propose to implement BER encoders and decoders on Very Large Scale Integration (VLSI) chips. A *VHDL* (Very high speed integrated circuit Hardware Description Language) model of these components shows that these are undoubtedly faster than the equivalent software architecture.

Chapter 23

ASN.1 and the formal languages SDL, TTCN, GDMO

Contents

Mr Le Hir was a scholar and a saint; he was both at the utmost. Within a single person, this cohabitation of two entities, which could hardly go together in general was taking place without much noticeable conflict: the saint always took over and ruled as a master.

Ernest Renan, *Childhood Memories.*

So far, ASN.1 has been described as a notation for modeling data transfer of telecommunication protocols. Part of notation is in fact also used within the formal specification language SDL, the tabular notation TTCN for protocol testing and the GDMO notation for network management. In this case, ASN.1 should be considered more as a typing language for the data handled by those three notations.

23.1 The formal specification language SDL

SDL[1] (Specification and Description Language) is a formal language for specifying telecommunication systems. It provides concepts for structuring such systems and defining their behavior as well as their data. It was first standardized in 1976 [Z.100]; the object paradigm together with a better modularity were introduced in 1992.

Originally designed for specifying signalling systems and the way they interwork, it was extended to take into account all the aspects of switching systems, protocols in general, telecommunication services, data processing, etc.. Its popularity has now grown beyond the telecommunication community particularly because of the graphical syntax SDL/GR. The graphical representation provides different items for every 'action' or event; it is used jointly with the textual syntax SDL/PR, which is now mainly a common format for the industrial tools associated with SDL.

An SDL system is a set of finite state machines that work in parallel and communicate with one another and the system's outside environment by means of messages (called 'signals'). Like all formal notations (i.e. with a formal semantic, which is not quite the case for ASN.1), SDL has the following advantages: it describes in a rigorous and structured way the different functionalities, it avoids ambiguities that may lead to divergent interpretations, it enables the designer to detect and to analyze errors early during conception, it improves mutual understanding between users and designers, it increases interworking and make maintenance and updating easier.

A valuable SDL software development kit should have at least the following features: a graphical editor, a semantic checker to detect inconsistencies in the specification, a *C*++ code generator to produce the procedures that implement the system whose behavior has been specified, and an automatic test suite generator[2].

If the data transfer (or signals) between the processes and the system's environment is specified in ASN.1, the user merely needs to compile the ASN.1 modules to generate the encoding and decoding procedures and is automatically provided with the whole communicating

[1] http://www.sdl-forum.org/SDL/index.htm, http://www.telelogic.se/solution/language/sdl.asp

[2] http://www.irisa.fr/EXTERNE/projet/pampa/VALIDATION/TGV/TGV.html, for example.

system (behavior *plus* data exchanges). The numerous assets of such an approach are fairly obvious.

The joint use of ASN.1 and SDL was ratified in 1995 by the Z.105 recommendation (it has further been split into two recommendations: [Z.105] and [Z.107]) , which allows ASN.1 definitions in SDL diagrams where ASN.1 modules can be imported[3]. This recommendation presents a consistent way of specifying the behavior of a telecommunication system: the structure and behavior of the system itself are described in SDL, the data and the signals are defined in ASN.1 and the data encoding refers to the associated ASN.1 encoding rules (see Part III).

ASN.1 is actually an alternative to *ACT ONE*[4], the default data type language for SDL, but tends more and more to take over it. *ACT ONE* is an algebra-based language using axioms to describe the type's properties but not how these can be obtained. In general , it is therefore hard to include the language as a whole in SDL related tools and in simulators in particular.

The document [Z.100S1] gives directions for using MSC (Message Sequence Chart) diagrams of messages with the ASN.1-typed SDL language. The MSC diagrams describe specific sequences of events (or 'stimuli') while SDL defines the behavior of every one of these stimuli in each of their possible states.

The [Z.105] recommendation[5] imposes a few restrictions on ASN.1 to be able to use in SDL the first part [ISO8824-1] of the ASN.1:1997 standard:

- the hyphens "-" used in ASN.1 references must be replaced by

[3]Except this [Z.105] recommendation (whose way of putting things is somewhat obscure), details of combined use of ASN.1 and SDL can be found in [OFM96, Appendix A], [Sch94], and in the numerous and strongly recommended documents available on the ETSI website like [ETSI114], [ETSI295], [ETSI298], [ETSI383] and [ETSI414].

[4]*ACT ONE* was at first adopted by the formal language LOTOS [ISO8807] as the data type language and then re-used by SDL in the middle of the 1980s when trying to bring into line the two languages.

[5]Before its 1999 edition, the SDL notation was not case-sensitive; hence the Z.105:1995 recommendation imposed also that ASN.1 definitions included directly in an SDL diagram must end with a semicolon to be supported by the notation (this rule does not apply to the definitions that appear *within* ASN.1 modules imported in an SDL diagram); as a result, two ASN.1 types could not have the same name regardless of the case whereas a value and a type could have the same labels since these two concepts are distinct in SDL (in 'pure' ASN.1 the two labels' initials should at least be case-different); it was, however, recommended to keep these distinct cases in SDL diagrams.

underscores "_" in SDL not to confuse them with the subtraction operator; if the definition of a type labelled `Date-and-time` is imported from an ASN.1 module, this type must be referenced by Date_and_time[6] in the SDL diagram;

- if external references such as `ModuleName.TypeReference` or `ModuleName.valueReference` appear in an SDL module, the dot "." should be nested in spaces (as in 'ModuleName . TypeReference') because SDL allows dots in references (as well as other symbols like curly or square brackets);

- tags are allowed but ignored by SDL; it is therefore recommended to insert the `AUTOMATIC TAGS` clause in the headers of ASN.1 modules.

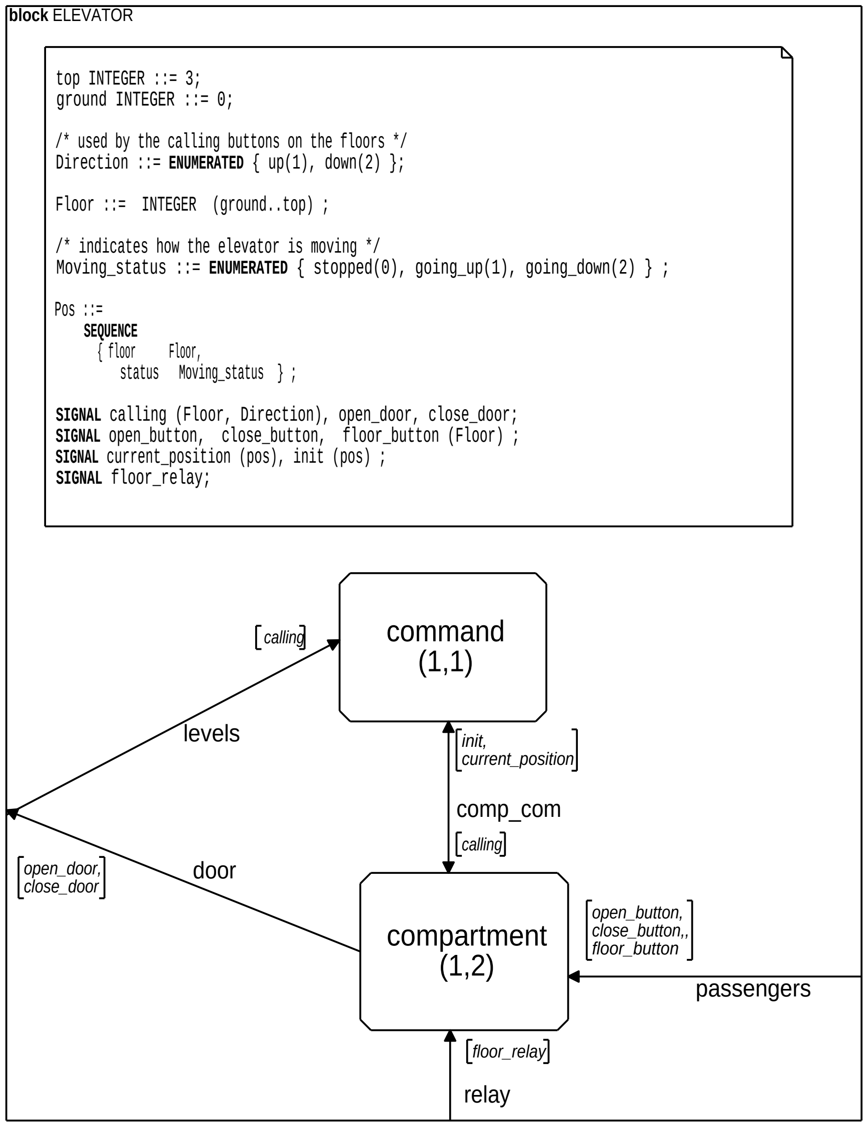
It is recommended to collect in a single ASN.1 module all the ASN.1 definitions imported in an SDL diagram from different ASN.1 modules (in part or as a whole using the `IMPORTS` or use clauses). This avoids to fall back on some ASN.1/SDL-mixed dialects for removing syntactic ambiguities. We would recommend to use two dedicated tools in parallel for each one of the two notations instead.

In addition, the [Z.105] recommendation enumerates all the possible operators applicable to data described in ASN.1; those are summarized in Appendix B on page 509. In fact the description of these operators is that which is already known for SDL since this recommendation merely translates every ASN.1 type into an *ACT ONE* type. These correspondences are gathered together into algebraic definitions in the devoted package called Predefined[7]. It is nevertheless recommended to use *ACT ONE* types only in other *ACT ONE* type definitions and reference ASN.1 types in other ASN.1 types rather than mixing the two.

The chart on Figure 23.1 on the next page models with SDL the architecture of an elevator with two compartments and a general control process. The latter receives messages from the various floors, determines a compartment that deals with every call and collects and manages information coming from the environment or the compartments (call, floor, current directions of the elevators).

[6]In the rest of this chapter as well as in Appendix B, the sans serif font is kept for the SDL notation while the ASN.1 notation remains in `teletype font`.

[7]This correspondence is also perfectly described in [OFM96], even though this book was published before the first edition of recommendation [Z.105].



```
top INTEGER ::= 3;
ground INTEGER ::= 0;

/* used by the calling buttons on the floors */
Direction ::= ENUMERATED { up(1), down(2) };

Floor ::=  INTEGER  (ground..top) ;

/* indicates how the elevator is moving */
Moving_status ::= ENUMERATED { stopped(0), going_up(1), going_down(2) } ;

Pos ::=
    SEQUENCE
      { floor     Floor,
         status   Moving_status  } ;

SIGNAL calling (Floor, Direction), open_door, close_door;
SIGNAL open_button,  close_button,  floor_button (Floor) ;
SIGNAL current_position (pos), init (pos) ;
SIGNAL floor_relay;
```

Figure 23.1: SDL interconnection diagram for an elevator

Today's studies of ITU-T and ETSI have led to the emergence of a new SDL recommendation called SDL-2000, which refers (and supports) the four parts[8] of the ASN.1:1997 standard. SDL has became case-sensitive and has a clause like 'USING ENCODING BER' to specify the transfer syntax of the messages. All the justifications for these points can be found in [ETSI680], [ETSI114] or [ETSI295]. The November 1999 version of recommendation [Z.105] is dedicated to the combination of SDL with ASN.1 modules (hence it forbids inserting ASN.1 definitions directly in SDL specifications) while a new [Z.107] recommendation allows to embed ASN.1 definitions in SDL.

23.2 The TTCN language for test suites

Testing is part and parcel of the development cycle of a telecommunication protocol. The use of a formal specification notation (like SDL and ASN.1 in particular) for telecommunication services or protocols allows using automated techniques for testing [BG94]. In 1983, ISO started working on the production of tests regardless of protocols and testing means. This lead to the ISO 9646 standard series and more particularly to a language for describing tests very much adapted to protocol testing: TTCN[9] (Tree and Tabular Combined Notation, [ISO9646-3]).

Contrary to a piece of software, the implementation of a protocol is tested as a 'black box' rather than following its behavior step by step (its existence is not even assumed): it is only considered at the interface level by sending messages and collecting answers. The test ensures that the implementation conforms to the specification so that it can interwork (under the same conditions as those of the tests) with any other system that uses the same protocol. The features and particular options of this implementation are specified in a *protocol instance compliance statement* (PICS, see footnote 4 on page 379).

The TTCN language offers standard templates for these test suites, called 'abstract' because they do not go into detail as far as the encoding and decoding are concerned but focus on the interactions by means of PDUs (or by using service primitives provided by the different layers of

[8]Originally, Part 2 and 3 were not included for lack of time because modeling them in *ACT ONE* proved difficult.

[9]http://www.telelogic.se/solution/language/ttcn.asp,
http://dis.sema.es/products/Concerto/TtcnPromo/Concerto_TTCN.html,
http://www-iiuf.unifr.ch/∼scheurer/ttcn.html

ASN.1 Type Constraint Declaration
Constraint Name: AcceptTypesDefCtr **ASN.1 Type**: AcceptTypes **Derivation Path**: **Comments**: default constraint for values of AcceptTypes
Constraint Value
{ standards ('10??'B, '01??'B) IF_PRESENT, others {} IF_PRESENT }
Detailed Comments:

Figure 23.2: A TTCN table

the OSI model). This test suite can be run several times or even re-executed later for subsequent takings resulting from the communicating system updating. Assisting tools for writing test suites can be found on the market and a TTCN compiler enables such tool to generate procedures for executing these suites. TTCN has been used in numerous domains such as the integrated services digital network (ISDN), GSM, the X.500 directory, the virtual terminal (VT) and file transfer FTAM (see on page 81), etc..

TTCN describes its test suites as hierarchical structures of tables (after which it is named), which indicate the whole set of possible events (sending or receiving a message, a timeout...) and returns a verdict depending on the events observed and their ordering. Every piece of information relevant to the test suite is described in a tabular notation as in Figure 23.2 (the type `AcceptTypes` is defined on page 415). This graphical representation TTCN.GR is equivalent to a textual syntax TTCN.MP (Machine Processable) where the rows and columns are replaced by keywords.

In order to use directly the ASN.1 specification of the protocol to be tested, TTCN includes a subset of the first part [ISO8824-1] of the ASN.1 standard (in fact quite close to ASN.1:1990). Even though ASN.1 is more popular among the specifiers of the highest layers of the OSI model, TTCN is also used for low-layer protocol test suites. In return, ASN.1 provides functionalities that TTCN on its own has not: `DEFAULT` and `OPTIONAL` clauses (by default, all the fields inserted in TTCN tables are optional), the `REAL`, `SET` and `SET OF` types, recursive types (forbidden in TTCN tables), subtype constraints.

Every ASN.1 type is either defined in a separate table entitled 'ASN.1 Type Definition' or imported from a module in a 'ASN.1 Type Definitions By Reference' table, which gathers together all these imported modules (since external references such as '`ModuleName.TypeName`' are forbidden in TTCN). When importing these modules the hyphens "`-`" are replaced by underscores "_" as in SDL.

The combined use of ASN.1 & TTCN and the description of TTCN functions that can be applied to ASN.1 values are perfectly described in [ETSI101], [BG94], [ETSI56] and [ETSI141]. The syntax of ASN.1 values is extended to use the wildcards '`?`' (which stands for any value), '`*`' (any value or none at all) and `OMIT` (no value).

ETSI and ISO have completed the next TTCN standard, called TTCN-3 or [Z.140]: it is fully compliant with ASN.1:1997, i.e. it includes information object classes, objects and object sets, extensibility (more specifically for tests involving PER encoding), the exception marker (which could be useful if recorded in the log files of TTCN tools) and parameterization.

23.3 The GDMO notation for network management

After the development of numerous telecommunication protocols and as networks were sprawling and tended to be more and more heterogeneous, technical supervising and administrating these protocols and networks became a necessity. The functional areas that are usually covered by supervision are configuration management (set-up, resource visualization, state and command management), security (authentication, authorization and accounting), fault (alarm supervision, fault location, tests and test measurement), performance (data collection, traffic management and quality of service) and accounting (account statement, invoicing parameters).

In order to take into account the heterogeneity of equipment and the different sizes of open networks (whether public or private), supervision[10] relies on a virtual representation of the network that offers high-level functions independent from the underlying equipment (ergo from its manufacturer) [Ram98].

[10]http://www.dkrz.de/~k202046/em/products/sem/Manuals/dev_guide/network.doc.html

```
coffee-machine MANAGED OBJECT CLASS
  DERIVED FROM "ITU-T Rec. X.721|ISO/IEC 10165-2:1992":top;
  CHARACTERIZED BY coffee-machine-pkg;
  CONDITIONAL PACKAGES
    coin-container-pkg PRESENT IF
      !the instance has the attribute coin-level!;
REGISTERED AS {iso member-body(2) f(250) type-org(1) ft(16)
  asn1-book(9) chapter23(6) managedObjectClass(3)
  coffee-machine(0)};

coffee-machine-pkg PACKAGE
  BEHAVIOUR coffee-machine-beh;
  ATTRIBUTES
    coffee-machine-id GET,
    "ITU-T Rec. X.721|ISO/IEC 10165-2:1992":operationalState GET,
    water-level GET,
    coffee-level GET,
    sugar-level GET;
  NOTIFICATIONS alarm-level, out-of-order;
REGISTERED AS {iso member-body(2) f(250) type-org(1) ft(16)
  asn1-book(9) chapter23(6) package(4) coffee-machine(0)};

coffee-level ATTRIBUTE
  WITH ATTRIBUTE SYNTAX CoffeeMachine.CoffeeLevel;
  MATCHES FOR EQUALITY, ORDERING;
REGISTERED AS {iso member-body(2) f(250) type-org(1) ft(16)
  asn1-book(9) chapter23(6) attribute(7) coffee-level(0)};

CoffeeMachine{iso member-body(2) f(250) type-org(1) ft(16)
  asn1-book(9) chapter23(6) asn1Module(2) coffee-machine(0)}
DEFINITIONS ::=
BEGIN
CoffeeLevel ::= INTEGER { empty(0), low(1), high(10) }
-- ...
END
```

Figure 23.3: Extract of the GDMO specification for supervising a coffee machine

The equipment interfaces and software components of the network are modeled by these objects (called *managed objects*), which contain only properties that are useful for supervision. These properties can be:

- attributes reflecting the state of the network resource, a type of service...;
- actions that the supervisor may take on the resource (performance tests for example);
- notifications sent by the resource when a problem occurs;
- behaviors explaining how the resource reacts on the management interface's level.

Every one of these properties is modeled by a *template* described in the semi-formal object notation GDMO (Guidelines for the Definition of Managed Objects) [ISO10165-4]. Indeed, the structure of the network and its equipment is more easily represented using the object paradigm. Figure 23.3 on the preceding page presents some GDMO templates for supervising a coffee machine [Heb95].

The GDMO templates[11] can reference ASN.1 types and values every time it is necessary. Definitions should conform to the ASN.1:1997 [ISO8824-1] standard, but before migrating modules from ASN.1:1990 to ASN.1:1994/97 (see Section 6.4.2 on page 73), one must check that the GDMO tools to be used support the functionalities introduced in 1994 such as information object classes and parameters.

We may then come across:

- ASN.1 external value references (see Section 9.3 on page 117) in the `DEFAULT VALUE` and `INITIAL VALUE` clauses of the `PACKAGE` templates to represent the initial and default value of an attribute;
- an external type reference in the `PERMITTED VALUES` and `REQUIRED VALUES` clauses of the `PACKAGE` templates to represent the permitted and required value sets of an attribute;

[11]It is interesting to note that, but for a few syntactic changes, every one of these templates could be represented by an information object class (see Chapter 15), which would make buying a GDMO *parser* unnecessary (even less so since it often supports the ASN.1 standard rather poorly).

- an external type reference in the `WITH ATTRIBUTE SYNTAX` clause of the `ATTRIBUTE` templates to indicate the ASN.1 type of the values that this attribute can take;

- an external type reference in the `WITH INFORMATION SYNTAX` and `WITH REPLY SYNTAX` clauses of the `ACTION` templates to indicate the arguments type of the action and the type of the data returned;

- an external type reference in the `WITH INFORMATION SYNTAX` and `WITH REPLY SYNTAX` clauses, and identifiers (lexical tokens beginning with a lower-case letter) in the `AND ATTRIBUTE IDS` clause of the `NOTIFICATION` templates;

- an external type reference in the `WITH SYNTAX` clause of the `PARAMETER` templates to indicate the ASN.1 type of this parameter;

- ASN.1 syntax in comments for the `PRESENT IF` clauses of the `MANAGED OBJECT CLASS` templates, or in `BEHAVIOUR` templates.

The type and value definitions used in a GDMO specification are therefore collected in one or several ASN.1 modules and every reference is systematically preceded by the module where it is defined.

Besides, all templates written in GDMO can be registered in the universal registration tree (see Section 10.8 on page 153, and more particularly Figure 10.4 on page 161) and their object identifiers which appear in curly brackets in a `REGISTERED AS` clause, will be carried by the Common Management Information Protocol (CMIP[12]) each time the supervisor wants to operate a remote control on the network element but also when a component sends an alarm notification for instance. Note that it is the static descriptions of the managed objects that are actually registered ('for ever') in the registration tree but not the objects themselves, which are created and deleted dynamically every time the network configuration is changed (the managed objects are identified by other mechanisms, independent from those which enable the designer to define the GDMO templates).

The standards for network management (ITU-T X.700 series) have been rewritten to take into account the latest ASN.1:1997 edition; they will be officially available by February 2000. The use of information

[12]The data transfer of the CMIP protocol are obviously specified in ASN.1.

object classes[13] should make the encoding of data involved in the X.700 recommandation series easier (since they avoid repeating the same object identifier); they should also improve the subtyping of attributes that are open types (in place of the obsolete `ANY` type) and replace the `ERROR` and `OPERATION` macros referenced in the CMIP protocol. Moreover, the information object classes (see Chapter 15) would permit the specifier to define dynamically the parameters' type in the `ACTION` and `NOTIFICATION` templates using dynamically-extensible object sets.

Until the management tools are updated, it is recommended to produce two 'equivalent' modules (with the same object identifiers): one in ASN.1:1990, another in ASN.1:1997. For doing so, one may refer to the recommendations given in [ISO10165-4, Amendment 2] or [ETSI295] (see also Section 6.4 on page 72).

[13]Note that the concepts of managed object class in GDMO and information object class in ASN.1 have nothing in common.

Chapter 24

Other abstract syntax notations

Contents

We can move from one language to another, but in doing so we accept new constraints and make new mistakes. We also adopt a different tone, enjoying the *je ne sais quoi* of *Sprachgefühl*.

Robert Darnton, *The Great Cat Massacre*.

There exist many abstract notations that can be compared with ASN.1 and even compete with it in some respects. We shall present the most widespread and underline their similarities and their differences. We will see that, compared with ASN.1, these generally suffer from a lack of generality in their modeling and abstraction potential (less types or less generic ones, no extensibility, no parameterization, less efficient encoding than the PER) regarding the diversity of system architectures.

As a matter of course, we eliminate the data type notation of programming languages (*C*, *Java*[1]...) because, even though these notations are independent from system architecture, the same does not go for the internal memory storage, which may change from one system to another thereby forbidding any common and generic representation.

24.1 Sun Microsystems' XDR notation

The remote procedure call (RPC) enables a machine (the client) to execute a procedure on another machine of a network (the server) which means that for the client application, executing a remote procedure is similar to calling a local subroutine [Cor91]. It is equivalent to the Remote Operation Service Element (ROSE) of OSI (see on page 80). RPC is the cornerstone of the distributed file system NFS (Network File System), which enables a user to manage files in a transparent way on different machine architectures; it is also the basis of the Network Information Service (NIS) directory, also known as the 'yellow pages'.

One of the first RPC mechanisms was developed in the 1970s by James White who would later designed the *Courier* notation, Xerox's RPC mechanism whose abstract notation was the forerunner of ASN.1 (see History on page 60). The RCP mechanism, which became *de facto* a standard for Unix systems was designed by Sun Microsystems Inc. in 1985.

To model the interface between client and server, viz. describe the data to be exchanged, Sun designed the XDR notation (eXternal Data Representation) [RFC1832]. Strongly influenced by *Courier*, XDR syntax is close to the *C* language's, which is very much to the liking of the communicating application programmers. One could even say that XDR is for *C* what *Courier* is for *Mesa*! (see footnote 12 on page 60.) From an XDR model of an interface, an RPCGEN compiler (many of the versions of which are available in the public domain) generates a '`.c`'

[1]Even though *Java* makes it possible to exchange structured data between a client and a server regardless of their architectures, these two applications need to be implemented in *Java*, whereas ASN.1 is independent from the programming language and can therefore make two applications communicate whether they are implemented in the same language or not. In addition, *Java* imposes size limitations on integers whereas ASN.1 imposes no such limitation. In Chapter 22, we mentioned that ASN.1 compilers for *Java* are being developed: they will provide *Java* with ASN.1 functionality and flexibility.

source file for the client, another '`.c`' file for the server and a common '`.h`' header file.

XDR was only thought up as a means of exchanging data in a simple and efficient way in an RPC-like architecture and cannot model easily such complex structures as those handled in the context of X.400 e-mail systems, which eventually lead to ASN.1 (see Section 7.2 on page 81): no mechanism exists for defining optional components in structures (but for using 'discriminated unions'); the mere notion of extensibility does not (and cannot) exist since the XDR description of types relates directly to their encoding; the concepts of module and imports are unknown to XDR so that specifying complex interfaces and re-using existing modules is more difficult; the absence of a type for bit strings is a weak point in a multimedia environment.

Besides, every time the communicating applications' architectures reach the limit of the XDR encoding, a new type has to be introduced in the notation and new encoding rules have to be associated with it: this happened for instance for the `hyper integer`, type which was specifically created so that 64-bit integers could be supported (incidentally there is no superlative left, so how is it going to keep up with the 128-bit integers?!).

Concerning the transfer syntax, all the values (including the booleans) are encoded on 32 bits. This fixed alignment prevents it from systematically encoding the value length. At first thought, the four-byte alignment, which is quite close to the machines' own architecture, could be expected to benefit from more efficient routines and therefore induce no translation at all. Several studies, however, point out that with very little effort put into optimizing[2] the ASN.1 compilers, BER (and even PER) encoders and decoders are often more efficient (regarding size and speed) than their XDR counterparts ([MC93], [DHV92], [HC92], [Hos93b], [SN93]). XDR is therefore only recommended for today's most common machine architectures because the data transfer induced minimizes the conversion to their own formats.

[2]The BER decoders are sometimes criticized for spending most of their computing time in extracting and checking the tag and length fields (T and L). The decoders can, however, be optimized for ignoring the fields that are assumed to be known.

24.2 Apollo Computer's NIDL notation

For the development tool kit of distributed applications of the Network Computing Architecture (NCA), Apollo Computer Inc., a subsidiary of Hewlett-Packard, developed the Network Interface Description Language (NIDL) and the encoding rules called Network Data Representation (NDR) to model data transfer using RPC ([PR89], [KDL87], [ISO11578]).

Besides being also in the public domain as its counterpart from Sun, NIDL very much looks like Sun's XDR notation in the sense that it is very close to *C* language. Its encoding rules, however, are not based on a neutral transfer format: no tag field is transmitted but it is replaced by an information byte that indicates the sending-machine architecture and its internal-representation conventions (decimal format, bit weights...) because the encoding varies with the sender so that the receiver has to operate the appropriate conversions. It is an example of symmetric communications presented in Figure 2.3(a) on page 12.

The main inconvenience of this approach was discussed in Chapter 2: the receiver needs several converters for a network of heterogeneous machine architectures; if the adequate configuration converter is not available to the receiver, then the closest one is used, which may lead to erroneous data decoding. Otherwise, the decoder of every machine in the network should be updated so that the new format could be properly converted. This is quite similar to what was described for the LWER encoding rules in Section 21.1 on page 454. The encoding routines are obviously easy to implement since they are close to the sender's internal-memory storing format.

The benchmarks of the NDR compare with those of XDR if the architectures of the sender and the receiver are different; they perform better if the architectures are identical. Generally speaking, they remain less efficient than a PER or BER encoding [SN93].

24.3 OMG IDL language for CORBA

The Common Object Request Broker Architecture (CORBA) is an object-oriented architecture specification. It is independent from the programming language, the platform implementation and the provider; it offers services like labelling or error notification [OPR96]. It was developed in 1990 by the Object Management Group (OMG), an association of about 500 members in charge of promoting the object-oriented techniques in view to integrating the existing applications in distributed environments. It is now standardized [ISO14750].

With every CORBA class or service, is associated an interface that defines, in particular, the object's attribute types, i.e. the parameter types to be provided by a requesting client, and the data types returned by the server as the result of this request. This interface is described in IDL (Interface Definition Language) whose syntax is close to *C*++. Independent from programming languages, the interface hides the implementation-specific details for every system architecture.

The interface definitions, once grouped in modules, are processed by an IDL compiler to generate communication procedures for both the client and the server (the use of an IDL compiler is very close to that of an ASN.1 compiler described in Figure 22.2 on page 465). For the moment, the target languages standardized by OMG are *C*, *C*++, *Smalltalk*, *Lisp*, *Java*, *Ada* and *COBOL*.

The Joint Inter-Domain Management (JIDM) forum created in 1993 by the Open Group (formerly X/Open Consortium), and TeleManagement Forum (formerly Network Management Forum) propose a bidirectional translation between GDMO/ASN.1 (see Section 23.3 on page 482) and IDL to allow the CORBA-compliant applications to support equipment that are usually managed by the CMIP protocol ([TMF96], [Gen96]).

The translation from IDL to ASN.1 poses no difficulties; it builds up a bridge from the OSI world to CORBA architectures to use automatic TTCN-test generation tools (see Section 23.2 on page 480), as explained in [ETSI56, Appendix B] for example. In the other way about, the translation suffers from loss of information (all tags, some subtype constraints, some abstract value definitions, some constructed type clauses for instance) because ASN.1 proves to be of a higher level than IDL. Moreover, when using BER encoding, the systematic use of tag and length fields allows the decoder to ignore the unrecognized components (for extensible structures in particular) while remaining capable of relaying those components the way it received them (see on page 250). Finally, a PER encoding is much less verbose than encodings produced with CORBA.

Since it is derived from the *C* language as XDR, IDL provides no generic integer type and imposes to restrict the length of the integer values by choosing among different pre-defined integer types. It does not allow for downwards referencing as it is frequently used in ASN.1 'top-down' specifications and does not support recursive types whose cycles are greater than 1. Finally, IDL syntax is not case-sensitive and imposes restrictions on the length of identifiers and references (which is restrictive for specification lengths). Renaming therefore also necessary to take into account the lexical differences between the two notations and the scope differences among identifiers.

24.4 RFC 822 notation for the Internet

The Augmented Backus-Naur Form (ABNF, [RFC2234]) can describe grammars by a syntax which software designers are already familiar with since it is very much like the syntax used for specifying the grammar of computing languages. Hence it is not particularly dedicated to the specification of data types.

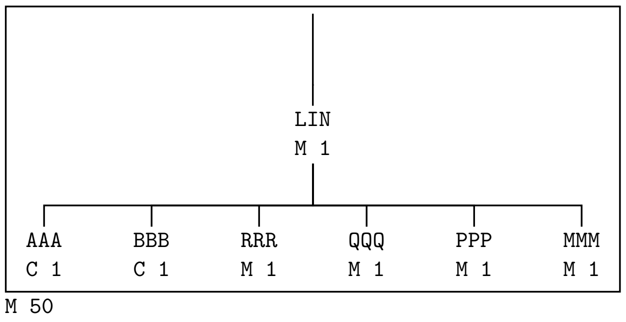
The [RFC822] encoding has been used for a very long time now for text-based Internet electronic mail. It is conceptually the simplest way of exchanging data between heterogeneous systems: these data are textually described in an alphabet that the systems[3] have agreed on. Simple, readable, extensible and easy to implement (if the extensions are sensible and homogeneous), this notation is, however, very expensive in terms of bandwidth and not appropriate for describing complex data structures.

24.5 EDIFACT notation

EDIFACT[4] (Electronic Data Interchange for Finance, Administration, Commerce, and Transport) is both a graphical notation and a set of encoding rules developed in parallel with the studies on the OSI model while people of one group were unaware of what the others were working on. Standardized by ISO [ISO9735], this notation was specifically designed for exchanging commercial data for which it ensures a unique capture and makes data processing easier.

[3]For example, the boolean value 'true' is represented by the character string "`TRUE`".

[4]http://www.unece.org/trade/untdid/welcome.htm



`M n` = Mandatory, from 1 to `n` repetitions
`C n` = Conditional, from 0 to `n` repetitions

Figure 24.1: A diagram of EDIFACT message

The EDIFACT exchange is a collection of messages (invoices for example) made of several segments (the invoice's lines) of data (quantity, address...). Every segment is labelled by a three-capital-letter name, e.g. `NAD` (Name And Address) for some individual's details, which has to be standardized in the Trade Data Elements Directory (TDED), an international directory [ISO7372]. The position of the data within the message pattern indicates the nature (and the content) of the transmitted data.

An example of EDIFACT graphical specification is given in Figure 24.1 (imagine it is the line of an ordering form). The tree-like structure introduces a hierarchy between the segments. The encoding is carried out by a depth-first traversal of this structure. By definition the EDIFACT standard uses a character-based encoding since the documents were originally meant to be sent by Telex. The different parts of a message are therefore delimited by specific symbols negotiated beforehand: by default the data components are separated by ":", the data themselves by "+" and a message ends with "'". It is therefore needless to specify the data lengths since all the data are delimited.

For an invoice, the following line:

Ref.	Description	Qty	Price (€)	Amount
5750 321Q	ASN.1 Compiler	1	15,000.00	15,000.00

can be encoded as:

```
LIN++5750321Q:CN1+1:21:PC+15000:CA:1+1+15000
```

This position-based encoding favors efficiency at the expense of flexibility and sometimes against syntactic rigor (some ambiguities in the graphical syntax were detected). The messages are thus designed with a structure similar to that of the documents they are meant to replace. All the delimiting symbols must therefore be transmitted even when the following item of data is absent not to shift the other data (in BER encoding the tag field T provides a means of determining this position). The messages and segments to be sent are necessarily pre-defined and encoded with a header that must be standardized in the TDED directory [ISO7372]. As a result, extensibility is not easily supported.

One can quickly be convinced that the EDIFACT notation is much less powerful[5] than ASN.1 simply by looking at Figure 24.1 on the page before; the former cannot be used for modeling protocol arbitrarily complex as those of the OSI Application layer: the translation of an EDIFACT diagram in ASN.1 is trivial but the reverse is not true [TH].

Using ASN.1 in this area would make it possible to take advantage of the secured mechanisms designed in the OSI world which prove very useful when exchanging digital data but would also come along with the numerous tools and compilers developed for ASN.1. The best way to convince oneself is to have a look at the user-friendly syntax of an information object class (see Section 15.3 on page 323) that preserves the notation's readability and is very much appreciated outside the computing-world.

Some EDI standards may also go towards XML (eXtensible Markup Language), a self-descriptive language derived from HTML which happens to be very much fashionable these days on the Web [Bra98]. Like EDIFACT, XML is more suitable for describing the logical structure of electronic documents and is therefore not appropriate for arbitrarily complex data.

[5] "*It would not be totally unreasonable to equate ASN.1 with Fortran and EDIFACT with COBOL!*" [Lar96].

Chapter 25

Epilogue

'And that', put in the Director sententiously, 'that is the secret of happiness and virtue – liking what you've *got* to do. All conditioning aims at that: making people like their unescapable social destiny.'

Aldous Huxley, *Brave New World*.

Part V

Appendices

Appendix A

Encoding/decoding simulations

We give the encoding/decoding of the value of type `GetRequest` that we used in Section 18.4 on page 415 and Section 20.7 on page 451. This simulation is generated automatically with the "`-test`" option of the ASN.1 compiler developed by OSS Nokalva[1].

It is divided into five stages: the value is encoded and then decoded in BER (using both the definite and the indefinite formats), in DER and finally in PER (using the aligned and the unaligned variants).

```
MyHTTP DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
GetRequest ::= SEQUENCE {
  header-only  BOOLEAN,
  lock         BOOLEAN,
  accept-types AcceptTypes,
  url          Url,
  ... }
AcceptTypes ::= SET {
  standards BIT STRING {html(0), plain-text(1), gif(2),
                        jpeg(3)} (SIZE (4)) OPTIONAL,
  others    SEQUENCE OF VisibleString (SIZE (4)) OPTIONAL }
Url ::= VisibleString (FROM ("a".."z"|"A".."Z"|"0".."9"|
                             "./-_~%#"))
END
```

[1] http://www.oss.com/products/tools.html

```
---------- Run 1 of the BER DEFINITE-Length Encoder ----------

        Unencoded PDU...

value GetRequest ::= {
  header-only TRUE,
  lock FALSE,
  accept-types {
    standards { html, plain-text } },
  url "www.asn1.com" }

        Checking constraints...
Constraints checking succeeded

   Tracing Information from the BER DEFINITE-Length Encoder...

GetRequest SEQUENCE: tag= [UNIVERSAL 16] constructed; length= 26
  header-only BOOLEAN: tag = [0] primitive; length = 1
    TRUE
  lock BOOLEAN: tag = [1] primitive; length = 1
    FALSE
  accept-types AcceptTypes SET: tag = [2] constructed; length = 4
    standards BIT STRING: tag = [0] primitive; length = 2
      0x04c0
  url Url VisibleString: tag = [3] primitive; length = 12
    "www.asn1.com"

PDU successfully encoded, in 28 bytes:
301A8001 FF810100 A2048002 04C0830C 7777772E 61736E31 2E636F6D

========== Run 1 of the BER Decoder for DEFINITE PDU ==========

        Tracing Information from Decoder...

GetRequest SEQUENCE: tag= [UNIVERSAL 16] constructed; length= 26
  header-only BOOLEAN: tag = [0] primitive; length = 1
    TRUE
  lock BOOLEAN: tag = [1] primitive; length = 1
    FALSE
  accept-types AcceptTypes SET: tag = [2] constructed; length = 4
    standards BIT STRING: tag = [0] primitive; length = 2
      0x04c0
  url Url VisibleString: tag = [3] primitive; length = 12
    "www.asn1.com"
```

```
PDU successfully decoded, in 32 bytes

        Decoded PDU...

value GetRequest ::= {
  header-only TRUE,
  lock FALSE,
  accept-types {
    standards { html, plain-text } },
  url "www.asn1.com" }

Copying the Decoded PDU...
Comparing the Original and Copied PDUs...
Value comparison succeeded

---------- Run 1 of the BER INDEFINITE-Length Encoder ----------

GetRequest SEQUENCE: tag= [UNIVERSAL 16] constructed; length= indef
  header-only BOOLEAN: tag = [0] primitive; length = 1
    TRUE
  lock BOOLEAN: tag = [1] primitive; length = 1
    FALSE
  accept-types AcceptTypes SET: tag= [2] constructed; length= indef
    standards BIT STRING: tag = [0] primitive; length = 2
      0x04c0
  EOC
  url Url VisibleString: tag = [3] primitive; length = 12
    "www.asn1.com"
EOC

PDU successfully encoded, in 32 bytes:
30808001 FF810100 A2808002 04C00000 830C7777 772E6173 6E312E63
6F6D0000
```

```
========== Run 1 of the BER Decoder for INDEFINITE PDU ==========

GetRequest SEQUENCE: tag= [UNIVERSAL 16] constructed; length= indef
  header-only BOOLEAN: tag = [0] primitive; length = 1
    TRUE
  lock BOOLEAN: tag = [1] primitive; length = 1
    FALSE
  accept-types AcceptTypes SET: tag= [2] constructed; length= indef
    standards BIT STRING: tag = [0] primitive; length = 2
      0x04c0
  EOC
  url Url VisibleString: tag = [3] primitive; length = 12
    "www.asn1.com"
EOC

PDU successfully decoded, in 32 bytes

---------- Run 1 of the DER Encoder ----------

GetRequest SEQUENCE: tag= [UNIVERSAL 16] constructed; length= 26
  header-only BOOLEAN: tag = [0] primitive; length = 1
    TRUE
  lock BOOLEAN: tag = [1] primitive; length = 1
    FALSE
  accept-types AcceptTypes SET: tag = [2] constructed; length = 4
    standards BIT STRING: tag = [0] primitive; length = 2
      0x06c0
  url Url VisibleString: tag = [3] primitive; length = 12
    "www.asn1.com"

PDU successfully encoded, in 28 bytes:
301A8001 FF810100 A2048002 06C0830C 7777772E 61736E31 2E636F6D

========== Run 1 of the DER Decoder ==========

GetRequest SEQUENCE: tag= [UNIVERSAL 16] constructed; length= 26
  header-only BOOLEAN: tag = [0] primitive; length = 1
    TRUE
  lock BOOLEAN: tag = [1] primitive; length = 1
    FALSE
  accept-types AcceptTypes SET: tag = [2] constructed; length = 4
    standards BIT STRING: tag = [0] primitive; length = 2
      0x06c0
  url Url VisibleString: tag = [3] primitive; length = 12
    "www.asn1.com"
```

```
PDU successfully decoded, in 32 bytes

---------- Run 1 of the PER ALIGNED Encoder ----------

        Unencoded PDU...

value GetRequest ::=
{
  header-only TRUE,
  lock FALSE,
  accept-types
  {
    standards { html, plain-text }
  },
  url "www.asn1.com"
}

        Checking constraints...
Constraints checking succeeded

        Tracing Information from the PER ALIGNED Encoder...

GetRequest SEQUENCE [fieldcount (not encoded) = 4]
  header-only BOOLEAN [length (not encoded) = 0.1]
    TRUE
  lock BOOLEAN [length (not encoded) = 0.1]
    FALSE
  accept-types AcceptTypes SET [fieldcount (not encoded) = 1]

    standards BIT STRING [length (not encoded) = 0.4]
      0xc0
  url Url VisibleString [length = 12.0]
    "www.asn1.com"
Total PDV length = 15.0

PDU successfully encoded, in 15 bytes:
56000C77 77772E61 736E312E 636F6D
```

The `ossPrintPer()` function used hereafter was developed specifically to give applications such as in protocol analyzers direct control over formatting and display of the output, while showing the fields of the PER output in great detail. In the majority of cases, direct control over the format of the output is not expected, in which case either the default display format can be used, or it can be altered simply by use of runtime flags.

The likes of "`OFFSET: 32,2`" means that the field occured in octet 32, at bit 2. The likes of "`LENGTH: 4,6`" means that the total length of the value that follows (including the "`length:`" field) is 4 octets and 6 bits longs. Symbols "`<`" and "`>`" such as in "`<000000>`" signify that the value is shown in binary (six `0` bits in this case). The period "`.`" in front of values such as "`.FE`" and "`<.10>`" signifies an octet boundary.

```
========== Run 1 of the PER ALIGNED ossPrintPER ==========

        ossPrintPER output...

value GetRequest ::=
{
  --TYPE INFORMATION: SEQUENCE
  --OFFSET: 0,0
  --extension flag: <.0>
  header-only TRUE,
    --TYPE INFORMATION: BOOLEAN
    --OFFSET: 0,1; LENGTH: 0,1
    --contents: <1>
  lock FALSE,
    --TYPE INFORMATION: BOOLEAN
    --OFFSET: 0,2; LENGTH: 0,1
    --contents: <0>
  accept-types
  {
    --TYPE INFORMATION: SET
    --OFFSET: 0,3
    --preamble: <10>
      --bit #0 = 1: 'standards' is present
      --bit #1 = 0: 'others' is absent
    standards { html, plain-text }
      --TYPE INFORMATION: BIT STRING (SIZE(4)) {html(0),
          plain-text(1),gif(2),jpeg(3)} OPTIONAL
      --OFFSET: 0,5; LENGTH: 0,4
      --contents: <110.0>
  },
  url "www.asn1.com"
    --TYPE INFORMATION: VisibleString
    --OFFSET: 1,1; LENGTH: 13,7
    --padding: <0000000>
    --length: .0C (decoded as 12)
    --contents: .77.77.77.2E.61.73.6E.31.2E.63.6F.6D
}
```

```
--TOTAL LENGTH: 15,0

         ossPrintPER finished...

========== Run 1 of the PER ALIGNED Decoder ==========

        Tracing Information from Decoder...

GetRequest SEQUENCE [fieldcount (not encoded) = 4]
  header-only BOOLEAN [length (not encoded) = 0.1]
    TRUE

  lock BOOLEAN [length (not encoded) = 0.1]
    FALSE
  accept-types AcceptTypes SET [fieldcount (not encoded) = 1]
    standards BIT STRING [length (not encoded) = 0.4]
      0xc0
  url Url VisibleString [length = 12.0]
    "www.asn1.com"
Total PDV length = 15.0

PDU successfully decoded, in 32 bytes

        Decoded PDU...

value GetRequest ::=
{
  header-only TRUE,
  lock FALSE,
  accept-types
  {
    standards { html, plain-text }
  },
  url "www.asn1.com"
}

Copying the Decoded PDU...
Comparing the Original and Copied PDUs...
Value comparison succeeded
```

```
---------- Run 1 of the PER UNALIGNED Encoder ----------

        Unencoded PDU...

value GetRequest ::=
{
  header-only TRUE,
  lock FALSE,
  accept-types
  {
    standards { html, plain-text }
  },
  url "www.asn1.com"
}

        Checking constraints...
Constraints checking succeeded

        Tracing Information from the PER UNALIGNED Encoder...

GetRequest SEQUENCE [fieldcount (not encoded) = 4]
  header-only BOOLEAN [length (not encoded) = 0.1]
    TRUE
  lock BOOLEAN [length (not encoded) = 0.1]
    FALSE
  accept-types AcceptTypes SET [fieldcount (not encoded) = 1]
    standards BIT STRING [length (not encoded) = 0.4]
      0xc0
  url Url VisibleString [length = 12.0]
    "www.asn1.com"
Total PDV length = 12.5

PDU successfully encoded, in 13 bytes:
560677EF DD761E7B 98AEC7BF 68
```

```
========== Run 1 of the PER UNALIGNED ossPrintPER ==========

        ossPrintPER output...

value GetRequest ::=
{
  --TYPE INFORMATION: SEQUENCE
  --OFFSET: 0,0
  --extension flag: <.0>
  header-only TRUE,
    --TYPE INFORMATION: BOOLEAN
    --OFFSET: 0,1; LENGTH: 0,1
    --contents: <1>
  lock FALSE,
    --TYPE INFORMATION: BOOLEAN
    --OFFSET: 0,2; LENGTH: 0,1
    --contents: <0>
  accept-types
  {
    --TYPE INFORMATION: SET
    --OFFSET: 0,3
    --preamble: <10>
      --bit #0 = 1: 'standards' is present
      --bit #1 = 0: 'others' is absent
    standards { html, plain-text }
      --TYPE INFORMATION: BIT STRING (SIZE(4)) {html(0),
          plain-text(1),gif(2),jpeg(3)} OPTIONAL
      --OFFSET: 0,5; LENGTH: 0,4
      --contents: <110.0>
  },
  url "www.asn1.com"
    --TYPE INFORMATION: VisibleString
    --OFFSET: 1,1; LENGTH: 11,4
    --length: <0000110.0> (decoded as 12)
    --contents: <1110111>.EF.DD.76.1E.7B.98.AE.C7.BF.<01101>
}
--PDU padding: <000>
--TOTAL LENGTH: 13,0

         ossPrintPER finished...

========== Run 1 of the PER UNALIGNED Decoder ==========

        Tracing Information from Decoder...
```

```
GetRequest SEQUENCE [fieldcount (not encoded) = 4]
  header-only BOOLEAN [length (not encoded) = 0.1]
    TRUE
  lock BOOLEAN [length (not encoded) = 0.1]
    FALSE
  accept-types AcceptTypes SET [fieldcount (not encoded) = 1]
    standards BIT STRING [length (not encoded) = 0.4]
      0xc0
  url Url VisibleString [length = 12.0]
    "www.asn1.com"
Total PDV length = 12.5

PDU successfully decoded, in 32 bytes

        Decoded PDU...

value GetRequest ::=
{
  header-only TRUE,
  lock FALSE,
  accept-types
  {
    standards { html, plain-text }
  },
  url "www.asn1.com"
}

Copying the Decoded PDU...
Comparing the Original and Copied PDUs...
Value comparison succeeded
All values encoded and decoded successfully.
```

Appendix B

Combined use of ASN.1 and SDL

Here are a few guidelines for combining ASN.1 types and values with SDL [Z.105]. Unfortunately, some tools do not support all of the following and impose some syntactic restrictions. These guidelines are consistent with the 1995 version of the [Z.105] recommendation. The new November 1999 version forbids inserting ASN.1 definitions directly in SDL specifications.

`BOOLEAN` : the operators available are: =, /=, not, and, or, xor, =>
- tip: rename the values `TRUE` and `FALSE` to make the specification more readable (with "Synonym yes BOOLEAN = TRUE;" for example);

`NULL` : the operators available are: =, /= (not much use!);

`INTEGER` : the operators available are: =, /=, >, <, >=, <=, Float, +, -, *, /, - (unary), rem, mod;
- all identifiers (beginning with a lower-case letter) appearing in definitions of type `INTEGER` and `BIT STRING` must be distinct within the same System, Block, Process, Procedure or Service; therefore the two following types cannot be defined within the same scope:

```
T1 ::= INTEGER {write(5)};
T2 ::= BIT STRING {write(0), read(1)};
```

- tip: use a value range constraint to make automatic test generation easier;

`ENUMERATED` : to every `ENUMERATED` it corresponds a specialization of the new type Enumeration introduced by [Z.105];
- operators: =, /=, >, <, >=, <=, Num, Pred, Succ, First, Last;
- examples: Day ::= ENUMERATED {monday(1), tuesday(2), wednesday(3), thursday(4), friday(5)};
Num(monday) = 1
Pred(monday) = error!
Last(wednesday) = friday
- note: extensibility is not supported;

`REAL` : a value is denoted either in the ASN.1:1990 form "{314,10,-2}" (without the identifiers `mantissa`, `base` and `exponent` introduced by ASN.1:1994), or in decimal form[1] "3.14";
- the particular values $+\infty$ et $-\infty$ (not recommended) are denoted PLUS_INFINITY and MINUS_INFINITY;
- {m, b, e} = m*Power(b,e);
- operators: =, /=, >, <, >=, <=, Float, +, -, *, /, - (unary), Power, Float, Fix, Exp;
- there exists no operator for changing only the mantissa, the base or the exponent;
- constraining this type is recommended to make automatic test generation easier;

`BIT STRING` : this type is equivalent to the new type bit_string of [Z.105] based on the generator String (with an index starting from 0) and the new type Bit;
- operators: =, /=; not, and, or, xor, => (bit-based logical operators); Length, MkString, //, First, Last, SubString; Bool, Octet String (coercion);
- examples: Rights ::= BIT STRING {read(0), write(1)};

myRights Rights ::= {read, write};	MkString(1) = '1'B
myRights(0) = 1 `-- equal`	'1'B // '01'B = '101'B
myRights(0) := 0; `-- changed`	Bool('1'B) = TRUE
myRights(user_read) := 0;	

- all identifiers (beginning with a lower-case letter) appearing in definitions of type `INTEGER` and `BIT STRING` must be distinct within the same System, Block, Process, Procedure or Service;

[1] This form will probably be added soon in the ASN.1 standard (see on page 143).

- tip: use a size constraint to make automatic test generation easier;

`OCTET STRING` : this type is equivalent to the new type octet_string of [Z.105] based on the generator String (with an index starting from 0) and the new type octet;
- operators: =, /=, Length, MkString, //, SubString, First, Last, Bit String, Octet String;
- tips: use a size constraint to make automatic test generation easier; use preferably a character string type whenever one of them is appropriate;

`OBJECT IDENTIFIER` : this type is equivalent to the new type Object_identifier of [Z.105] based on the generator String (with an index starting from 1) and the new type Object_element = INTEGER (0..MAX);
- for every identifier there must be an associated synonym of the form "Synonym identifier Object_element = number;"
- the standardized identifiers (in colorboxyellowyellow in Figure 10.4 on page 161) should be defined in a specific package since they are not part of the Predefined package in [Z.105];
- operators: =, /=, Length, MkString, //, SubString;

`SEQUENCE`, `SET` : from SDL viewpoint, these two types are equivalent to the constructor struct (but are encoded differently with the BER for instance);
- operators: =, /=, Extract, Present, Modify;
- examples:

```
Wife ::= SET { name      PrintableString,
               firstname PrintableString OPTIONAL,
               age       INTEGER DEFAULT 20 };
wife Wife ::= {name "Smith"}; | firstnamePresent(wife) = FALSE
wife!firstname = error!       | nameModify(wife, "Green")
nameExtract(wife) = "Smith"   |
```

- when defining a value, if there is an ambiguity on the type, the value must be prefixed by the notation <<type T>>;
- if the (. .) notation of *ACT ONE* is used, the fields of the value of type `SEQUENCE` must be defined in the alphabetical order; it is therefore recommended to put the values in curly brackets as in ASN.1;
- note: extensibility is not supported;

`SEQUENCE OF` : is equivalent to the generator String of *ACT ONE* (with an index starting at 1);
- operators: =, /=, Length, MkString, //, SubString, First, Last;
- examples: myPariTierce SEQUENCE (SIZE (3)) OF INTEGER ::= {5, 2, 12}; myPariTierce(2) := 7;
- tip: use a size constraint to make automatic test generation easier;

`SET OF` : is not equivalent to the generator Powerset of *ACT ONE* because the same value may appear several times in the multi-set; it is equivalent to the new generator Bag of [Z.105];
- operators: =, /=, > (strict superset), >=, <, <=, and (intersection), or (union), in, Length, MakeBag, Incl (element insertion), Decl, Take;
- examples for the type SEQUENCE OF INTEGER:

{1, 2} = {2, 1}	7 in {4, 7} = TRUE
{3, 3} /= {3}	Incl(5, {2, 3}) = {5, 2, 3}
{3, 3} > {3}	Take({1, 2, 3}) = 1
{1, 1} and {4, 1} = {1}	Take({ }) = error!

- tip: use a size constraint to make automatic test generation easier;

`CHOICE` : for every `CHOICE` type there is a newtype in SDL that includes the same operators as the struct constructor, as well as a Make constructor for every alternative of the `CHOICE`;
- operators: =, /=, Make, !present, Present, Extract, Modify;
- examples: Afters ::= CHOICE {cheese IA5String,
dessert IA5String};
mine Afters ::= dessert:"profiteroles";
dessertMake("profiteroles") = dessert:"profiteroles"
dessertExtract(mine) = "profiteroles"
dessertPresent(mine) = TRUE
mine!present = dessert
mine := cheeseModify(mine, "camembert");
- if there is an ambiguity on the value definition, we write <<type Afters>>dessert:"sabayon";
- tips: writing a `CHOICE` type is easier than the equivalent in *ACT ONE*; it is recommended to test the alternative retained with the !present operator before using it;

- notes: extensibility is not supported; the ASN.1 selection operator "<" is supported;

`ANY` : though no longer present in the [ISO8824-1] standard, this old ASN.1:1990 type remains in the [Z.105] recommendation, which does not include the notions of information object class and open type;
- this type must be used only for specifying signals still under construction or implementation parameters; in both cases, abstract values for this type cannot be defined because no value notation is provided by [Z.105]; a system may receive values of type `ANY` from the environment but cannot modify them;
- tip: use a `CHOICE` type instead, which will collect the different possible types [Z.105, Appendix 2, clause 4];
- note: this type with an infinite number of possible values is not appropriate for automatic test generation;

character string types : these types are equivalent to syntypes of the type CharString in SDL;
- the `UTF8String` type is not taken over by [Z.105] since it is defined in an amendment to the ASN.1:1994 standard;
- character strings can be denoted in quotes or double quotes; a string that contains a single character which is an SDL operator ("*") is denoted in quotes ('*') to prevent ambiguities;
- operators: =, /=; <, <=, >, >= (lexical order); Length, MkString, SubString, First, Last, Num;
- tip: use a size constraint to make automatic test generation easier;

`EXTERNAL` : all the operators of `SEQUENCE` type can be used;

subtyping : it is recommended to apply a subtype constraint each time it is possible to make automatic test generation easier;
- the value range symbol ".." can be replaced by ":" (if there is an ambiguity because one of the boundaries is a `CHOICE` value, it must be nested in round brackets); the union symbol "—" can be replaced by ",";
- note: extensibility of subtype constraints is not supported.

Abbreviations

3GPP	Third Generation Partnership Project
ACSE	Association Control Service Element [ISO8650-1]
AFNOR	Association Française de NORmalisation
ANSI	American National Standards Institute
APDU	Application Protocol Data Unit
ASCII	American Standard Code for Information Interchange
ASHRAE	American Society of Heating, Refrigerating and Air-conditioning Engineers
ASN.1	Abstract Syntax Notation One [ISO8824-1, ISO8824-2, ISO8824-3, ISO8824-4]
ATM	Asynchronous Transfer Mode
ATN	Aeronautical Telecommunication Network
BACnet	Building Automation and Control networks [Ash95]
BER	Basic Encoding Rules [ISO8825-1]
BMP	Basic Multilingual Plane
BNF	Backus-Naur Form
BSI	British Standards Institute
CAPTAIN	Character And Pattern Telephone Access Information Network System
CCITT	Consultative Committee on International Telephony and Telegraphy
CCR	Commitment, Concurrency, and Recovery [ISO9805-1]

CEN	European Committee for Standardization
CEPT	Conference of European Postal and Telecommunications Administrations
CER	Canonical Encoding Rules [ISO8825-1]
CMIP	Common Management Information Protocol [ISO9596-1]
CORBA	Common Object Request Broker Architecture
DER	Distinguished Encoding Rules [ISO8825-1]
DIN	Deutsches Institut fÆr Normung
DIS	Draft International Standard (ISO)
DNIC	Data Network Identification Code
DP	Draft Proposal (ISO)
DTAM	Document Transfer And Manipulation [T.433]
EBCDIC	Extended Binary Coded Decimal Interchange Code
ECMA	European Computer Manufacturers Association
EDI	Electronic Data Interchange
EDIFACT	Electronic Data Interchange for Finance, Administration, Commerce, and Transport [ISO9735]
ETSI	European Telecommunications Standards Institute
FPDAM	Final Proposed Draft AMendment
FTAM	File Transfer, Access, and Management [ISO8571-4]
GDMO	Guidelines for the Definition of Managed Objects [ISO10165-4]
GSM	Global System for Mobile communications
HTML	HyperText Markup Language
IA5	International Alphabet number 5
ICD	International Code Designator [ISO6523]
IDL	Interface Definition Language
IEC	International Electrotechnical Commission
IETF	Internet Engineering Task Force

INAP	Intelligent Network Application Protocol
INRIA	Institut National de Recherche en Informatique et Automatique (*French National Institute for Research in Computer Science and Control*)
IS	International Standard (ISO)
ISDN	Integrated Services Digital Network
ISUP	ISDN User Part of CCITT Signalling System No. 7
ISO	International organization for standardization
ISP	International Standardized Profile
ITU	International Telecommunications Union
ITU-T	International Telecommunications Union - Telecommunication standardization sector
JIDM	Joint Inter-Domain Management
JISC	Japanese Industrial Standards Committee
JTC	Joint Technical Committee (ISO/IEC)
LWER	LightWeight Encoding Rules
MAP	Mobile Application Part
MBER	Minimum Bit Encoding Rules
MHS	Message Handling Systems ('e-mail') [X.400]
MIT	Massachusetts Institute of Technology
MMI	Man-Machine Interface
MMS	Manufacturing Message Specification
MSC	Message Sequence Chart
NAPLPS	North American Presentation Level Protocol Syntax
NB	National Body (ISO)
NCA	Network Computing Architecture
NCBI	National Center for Biotechnology Information
NDR	Network Data Representation
NEMA	National Electrical Manufacturers Association
NFS	Network File System

NIDL	Network Interface Description Language
NIS	Network Information Service (‘yellow pages’)
NMF	Network Management Forum
NR	Numerical Representation
NTCIP	National Transportation Communications for Intelligent transportation systems Protocol
ODA	Office Document Architecture
ODIF	Office Document Interchange Format
ODMA	Open Distributed Management Architecture
ODP	Open Distributed Processing
OER	Octet Encoding Rules
OID	Object IDentifier
OMG	Object Management Group
OSI	Open Systems Interconnection
PDAM	Proposed Draft AMendment
PDU	Protocol Data Unit
PDV	Protocol Data Value
PER	Packed Encoding Rules [ISO8825-2]
PICS	Protocol Implementation Conformance Statement
PKCS	Public Key Cryptography Standards
PPDU	Presentation Protocol Data Unit
PPDV	Presentation Protocol Data Value
PTT	Post, Telephone and Telegraph administration
RFC	Request For Comments
RFID	Radio-Frequency IDentification
ROSE	Remote Operations Service Element [ISO9072-2]
RPC	Remote Procedure Call
RPOA	Registered Private Operating Authorities
RTSE	Reliable Transfer Service Element [ISO9066-2]

SC	SubCommittee (ISO)
SCCP	Signalling Connection Control Part
SDL	Specification and Description Language [Z.100]
SER	Signalling specific Encoding Rules
SET	Secured Electronic Transaction
SG	Study Group (ITU-T)
SGML	Standard Generalized Markup Language
SNMP	Simple Network Management Protocol
STMP	Simple Transportation Management Protocol
TC	Technical Committee (ISO)
TCP/IP	Transmission Control Protocol/Internet Protocol
TDED	Trade Data Elements Directory
TLV	Type-Length-Value or Tag-Lenth-Value
TTCN	Tree and Tabular Combined Notation [ISO9646-3]
UCS	Universal multiple-octet coded Character Set ([ISO10646-1], [Uni96])
UCS-2	Universal Character Set coded in 2 octets
UCS-4	Universal Character Set coded in 4 octets
UMTS	Universal Mobile Telecommunications Service
UTC	Universal Time Coordinated
UTF-8	UCS Transformation Format, 8-bit form
VLSI	Very Large Scale Integration
WAIS	Wide Area Information Server
WAP	Wireless Application Protocol
WG	Working Group (ISO)
WP	Working Party (ITU-T)
XDR	eXternal Data Representation
XER	XML Encoded Rules
XML	eXtensible Markup Language

XNS	Xerox Network Services

Bibliography

[Ash95] ASHRAE (American Society of Heating, Refrigerating and Air-conditioning Engineers). – *BACnet: A Data Communication Protocol for Building Automation and Control Networks.* – Standard 135-1995, 1995. ISSN: 1041-2336, http://www.ashrae.org/BOOK/bookshop.htm. 453, 513

[ASU86] Aho (A.), Sethi (R.) and Ullman (J.). – *Compilers: Principles, Techniques, and Tools.* – Addison-Wesley, 1986. http://cseng.awl.com/bookdetail.qry?ISBN=0-201-10088-6&ptype=0. 13, 98, 100, 108, 131, 295, 374, 464, 467

[Bel67] Bell (M.). – *Visible Speech: the Science of Universal Alphabetics; or Self-Interpreting Physiological letters, for the Writing of all Languages in one Alphabet.* – N. Trübner & Co, 1867. 3

[BG94] Baumgarten (B.) and Giessler (A.). – *OSI Conformance Testing Methodology and TTCN.* – North-Holland, Elsevier, 1994. ISBN: 0-444-89712-7, http://www.elsevier.nl/inca/publications/store/5/2/4/7/0/6/. 478, 480

[Bla91] Black (U.). – *OSI. A model for Computer Communications Standards.* – Prentice-Hall, 1991. ISBN: 0-13-637133-7. 20

[Bla96] Blackwell (T.). – Fast Decoding of Tagged Message Formats. *In: IEEE INFOCOM, Conference on Computer Communications*, ed. by IEEE Computer Society Press, pp. 224–231. – http://www.eecs.harvard.edu/~tlb/Infocom96.ps. 470

[Bra98] Bradley (N.). – *The XML Companion.* – Addison-Wesley, 1998. 492

[Bro76] Brooks (J.). – *Telephone: the First Hundred Years. The Wondrous Invention that Changed a World and Spawned a Corporate Giant.* – Harper & Row, 1976. ISBN: 0-06-010540-2. 5

[BS92] Bever (M.) and Schäffer (U.). – Coding Rules for High Speed Networks. *In: IFIP Conference on Upper Layer Protocol, Architectures and Applications*, ed. by Neufeld (G.) and Plattner (B.), Elsevier Science Publishers B.V. – http://www.eecs.harvard.edu/cgi-bin/selectrefs.pl?KEYS==BEVER92. 452

[BS93] Bilgic (M.) and Sarikaya (B.). – Performance Comparison of ASN.1 Encoders/Decoders Using FTAM. *Computer Communications*, vol. 16, no. 4, April 1993, pp. 229–240. – http://www.eecs.harvard.edu/cgi-bin/selectrefs.pl?KEYS==BILGIC93. 471

[Cha92] Chatras (B.). – *Méthode de spécification des protocoles de signalisation en ASN.1.* – Internal Report DE/CER/SSR/1359/BC, France Télécom R&D, March 1992. Only in French. 455

[Cha96] Chadwick (D.). – *Understanding OSI: The Directory.* – Chapman and Hall, 1996. ISBN: 1-85032-281-3, http://www.salford.ac.uk/its024/Version.Web/Contents.htm. 84, 328, 329

[Cha97a] Chahuneau (F.). – The Unicode Standard. A global Solution to Localization Problems in Electronic Documents. *Document numérique*, vol. 1, no. 4, December 1997, pp. 385–401. 183

[Cha97b] Chatras (B.). – *Spécification des règles de codage spécifiques pour la signalisation (SER).* – Internal Report NT/DAC/SSR/12, France Télécom R&D, July 1997. Only in French. 455

[Coh81] Cohen (D.). – On Holy Wars and a Plea for Peace. *IEEE Computer Magazine*, October 1981, pp. 48–54. 8

[Cor91] Corbin (J. R.). – *The Art of Distributed Applications: programming techniques for remote procedure calls.* – Springer-Verlag, 1991. ISBN: 0-387-97247-1. 486

[DHV92] Dabbous (W.), Huitema (C.), Vidaller Siso (L.), Joaquin (S.) and Berrocal (J.). – *Applicability of the Session and the Presentation Layers for the Support of High Speed Applications.* – Internal Report 144, INRIA Sophia Antipolis, October 1992. ftp://ftp.inria.fr/INRIA/publication/RT/RT-0144.ps.gz. 452, 453, 487

[ETSI56] ETSI - Signalling Protocols and Switching (SPS) – *Guide for the Use of Second Edition of TTCN (Revised Version).* – Draft Guide DEG/MTS-56, October 1998, ed. 0.0.2. 480, 489

[ETSI60] ETSI - Signalling Protocols and Switching (SPS) – *Guidelines for Using Abstract Syntax Notation One (ASN.1) in Telecommunication Application Protocols.* – Technical Report ETR 60, September 1995, 2nd ed. http://webapp.etsi.org/Publications/home.asp?wki_id=2115. 300

[ETSI101] ETSI - Methods for Testing And Specification (MTS) – *TTCN Interim Version Including ASN.1 1994 Support [ISO9646-3] (Second Edition Mock-Up for JTC 1/SC 21 Review).* – Technical Report TR 101 101, August 1998, ed. 1.1.1. http://webapp.etsi.org/Publications/home.asp?wki_id=4893. 480

[ETSI114] ETSI - Methods for Testing And Specification (MTS) – *Analysis of the use of ASN.1 94 with TTCN and SDL in ETSI deliverables.* – Technical Report ETR 101 114, November 1997. http://webapp.etsi.org/Publications/home.asp?wki_id=5377. 226, 475, 478

[ETSI141] ETSI - Methods for Testing And Specification (MTS) – *Protocol and Profile Conformance Testing Specifications: The Tree and Tabular Combined Notation (TTCN) Style Guide.* – Technical Report ETR 141, October 1994. http://webapp.etsi.org/Publications/home.asp?wki_id=2752. 480

[ETSI295] ETSI - Methods for Testing And Specification (MTS) – *Rules for the Transformation of ASN.1 Definitions Using X.681, X.682 and X.683 to Equivalent X.680 Construct.* – Technical Report TR 101 295, September 1998, ed. 1.1.1. http://webapp.etsi.org/Publications/home.asp?wki_id=5378. 77, 475, 478, 484

[ETSI298] ETSI - Signalling Protocols and Switching (SPS) – *Specification of protocols and services; Handbook for SDL, ASN.1 and MSC Development.* – Technical Report ETR 298, September 1996. http://webapp.etsi.org/Publications/home.asp?wki_id=3206. 475

[ETSI383] ETSI - Methods for Testing And Specification (MTS) – *Use of SDL in European Telecommunication Standards; Guidelines for facilitating validation and the development of conformance tests.* – Guide EG 201 383, October 1998, ed. 1.1.1. http://webapp.etsi.org/Publications/home.asp?wki_id=5781. 475

[ETSI414] ETSI - Signalling Protocols and Switching (SPS) – *Use of SDL in European Telecommunication Standards; Rules for testability and facilitating validation.* – European Telecommunication Standard ETS 300 414, December 1995. http://webapp.etsi.org/Publications/home.asp?wki_id=310. 475

[ETSI680] ETSI - Methods for Testing And Specification (MTS) – *A Harmonised Integration of ASN.1, TTCN and SDL.* – Technical Report TR 101 680, May 1999, ed. 1.1.1. http://webapp.etsi.org/Publications/home.asp?wki_id=5873. 478

[FDD96] Fouquart (P.), Dubuisson (O.) and Duwez (F.). – *Une analyse syntaxique d'ASN.1:1994*. – Internal Report RP/LAA/EIA/83, France Télécom R&D, March 1996. Only in French. 468

[Gen96] Genilloud (G.). – *Towards a Distributed Architecture for Systems Management*. – PhD Thesis, École Polytechnique Fédérale de Lausanne, 1996. http://icawww.epfl.ch/genilloud/Publications/FullThesis.pdf.gz. 489

[HC92] Huitema (C.) and Chave (G.). – Measuring the Performances of an ASN.1 Compiler. *In: IFIP Conference on Upper Layer Protocols, Architectures and Applications*, ed. by Neufeld (G.) and Plattner (B.), Elsevier Science Publishers B.V., pp. 105–118. 487

[HD98] Hétault (P.-M.) and Dubuisson (O.). – *Comparaison syntaxique de deux spécifications formelles ASN.1*. – Internal Report, France Télécom R&D, August 1998. Only in French. 469

[Heb95] Hebrawai (B.). – *GDMO. Object Modelling & Definition for Network Management*. – Technology Appraisals, 1995. ISBN: 1-871802-30X. 482

[Hor96] Horrocks (J.). – *European Guide to Telecommunications Standards*. – Horrocks Technology, 1996. ISBN: 1-900015-06-4 & 1-900015-05-6. 57, 59

[Hos93a] Hoschka (P.). – Towards Tailoring Protocols to Application Specific Requirements. *In: IEEE INFOCOM, Conference on Computer Communications, San Francisco*, pp. 647–653. – http://www.inria.fr/rodeo/personnel/hoschka/93infocom.paper.ps.gz, http://www.inria.fr/rodeo/personnel/hoschka/93infocom.slides.ps.gz. 470

[Hos93b] Hoschka (P.). – Use of ASN.1 for Remote Procedure Call Interfaces (*slides*). *In: Interop Europe, Paris*. – http://www.inria.fr/rodeo/personnel/hoschka/93interop.slides.ps.gz. 487

[Hos96] Hoschka (P.). – Automatic Performance Optimisation by Heuristic Analysis of a Formal Specification. *In: Formal Description Techniques IX, Kaiserslautern*, ed. by Gotzheim (R.) and Bredereke (J.), pp. 77–92. – http://www.inria.fr/rodeo/personnel/hoschka/96FORTE.paper.ps.gz. 470

[Hos97] Hoschka (P.). – *Compact and Efficient Presentation Conversion Routines.* – Internal Report RR 3291, INRIA Sophia Antipolis, October 1997. ftp://ftp.inria.fr/INRIA/publication/RR/RR-3291.ps.gz. 470

[HSO94] Hart (K. W.), Searls (D. B.) and Overton (G.). – SORTEZ: a Relational Translator for NBCI's ASN.1 Database. *CABIOS*, vol. 10, no. 4, 1994, pp. 369–378. 470

[HTN] Harumoto (K.), Tsukamoto (M.) and Nishio (S.). – *Design and Implementation of the Compiler for an ASN.1 Database on OODBMS.* – Department of Information Systems Engineering, Osaka University, Japon. 470

[Hui90] Huitema (C.). – *Definition of the Flat Tree Light Weight Syntax (FTLWS).* – Internal Report, INRIA Sophia Antipolis, July 1990. 452

[ISO646] *Information Technology - ISO 7-bit coded character set for information interchange.* – International Standard ISO/IEC 646:1991. http://www.iso.ch/cate/d4777.html. 172, 176, 193, 195, 442

[ISO2022] *Information technology – Character code structure and extension techniques.* – International Standard ISO/IEC 2022:1994. http://www.iso.ch/cate/d22747.html. 177, 182, 197, 406

[ISO2375] *Data processing – Procedure for registration of escape sequences.* – International Standard ISO 2375:1985. http://www.iso.ch/cate/d7217.html. 180, 182, 406

[ISO3166-1] *Codes for the representation of names of countries and their subdivisions - Part 1: Country codes.* – International Standard ISO/IEC

3166-1:1997. http://www.iso.ch/cate/d24591.html, ftp://ftp.isi.edu/in-notes/iana/assignments/country-codes. 158

[ISO6093] *Information processing - Representation of numerical values in character strings for information interchange.* – International Standard ISO/IEC 6093:1985. http://www.iso.ch/cate/d12285.html. 142, 400, 401, 420

[ISO6523] *Information technology - Structure for the identification of organizations and organization parts - Part 1: Identification of organization identification schemes. Part 2: Registration of organization identification schemes.* – Draft International Standard ISO/IEC 6523. http://www.iso.ch/cate/d25773.html, http://www.iso.ch/cate/d25774.html. 158, 514

[ISO6937] *Information technology – Coded graphic character set for text communication – Latin alphabet.* – International Standard ISO/IEC 6937:1994. http://www.iso.ch/cate/d13465.html. 179

[ISO7372] *Trade data interchange – Trade data elements directory (Endorsement of UNECE/TDED, volume 1).* – International Standard ISO/IEC 7372:1993. http://www.iso.ch/cate/d23441.html. 491, 492

[ISO7498-1] *Information technology - Open Systems Interconnection - Basic reference model: The basic model.* – International Standard ITU-T Rec. X.200 (1994) | ISO/IEC 7498-1:1994. http://www.itu.ch/itudoc/itu-t/rec/x/x200-499/x200.html. 18

[ISO7498-3] *Data networks and open system communication OSI networking and system aspects - Naming, Addressing and Registration.* – International Standard ITU-T Rec. X.650 (1996) | ISO/IEC 7498-3:1997. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x650.html. 154

[ISO8571-4] *Information processing systems - Open Systems Interconnection - File Transfer, Access and Management - Part*

4: File Protocol Specification (+ Technical Corrigendum 1:1992 + Amendment 1:1992: Filestore Management + Amendment 2:1993: Overlapped access + Amendment 3: Service enhancement + Amendment 4:1992 + Technical Corrigendum 1:1995). – International Standard ISO/IEC 8571-4:1988. http://www.iso.ch/cate/d15851.html. 81, 514

[ISO8601] *Data elements and interchange formats - Information Interchange - Representation of dates and times (+ Technical corrigendum 1:1991).* – International Standard ISO/IEC 8601 (1988). http://www.iso.ch/markete/moreend.htm, http://www.ft.uni-erlangen.de/~mskuhn/iso-time.html, ftp://ftp.uni-erlangen.de/pub/doc/ISO/ISO8601.ps.Z. 199, 201

[ISO8650-1] *Information technology – Open Systems Interconnection – Connection-oriented protocol for the Association Control Service Element: Protocol specification (+ Amendment1: Incorporation of extensibility markers).* – International Standard ITU-T Rec. X.227 (1995) | ISO/IEC 8650-1:1996. http://www.itu.ch/itudoc/itu-t/rec/x/x200-499/x227.html. 25, 77, 80, 135, 241, 299, 356, 513

[ISO8807] *Information processing systems - Open Systems Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour.* – International Standard ISO/IEC 8807 (1989). http://www.iso.ch/cate/d16258.html. 475

[ISO8822] *Information technology - Open Systems Interconnection - Presentation service definition.* – International Standard ITU-T Rec. X.216 (1994) | ISO/IEC 8822:1994. http://www.itu.ch/itudoc/itu-t/rec/x/x200-499/x216.html. 15, 20, 154

[ISO8823-1] *Information technology - Open Systems Interconnection - Connection-oriented presentation protocol: Protocol specification.* – International Standard ITU-T Rec. X.226 (1994) | ISO/IEC 8823-1:1994. http://www.itu.ch/itudoc/itu-t/rec/x/x200-499/x226.html. 20, 77, 80, 111, 132, 154, 244, 295, 361, 414, 452

[ISO8824-1] *Information Technology - Abstract Syntax Notation One (ASN.1): Specification of Basic Notation.* – International Standard ITU-T Rec. X.680 (1997) | ISO/IEC 8824-1:1998. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x680.html. 69, 73, 76, 136, 185, 197, 244, 252, 475, 479, 482, 511, 513

[ISO8824-1Amd1] *Information Technology - ASN.1: Specification of Basic Notation - Amendment 1: Relative Object Identifiers.* – Amendment ITU-T Rec. X.680 (1997)/Amd.1 (1999) | ISO/IEC 8824-1:1998/Amd.1:1999. http://www.itu.int/itudoc/itu-t/approved/x/x680amd1.html. 167, 169

[ISO8824-1Amd2] *Information Technology - ASN.1: Specification of Basic Notation - Amendment 2: ASN.1 Semantic model.* – Amendment ITU-T Rec. X.680 (1997)/Amd.2 (1999) | ISO/IEC 8824-1:1998/Amd.2:1999. http://www.itu.int/itudoc/itu-t/approved/x/x680amd2.html. 121, 197

[ISO8824-1TC1] *Information Technology - ASN.1: Specification of Basic Notation - Technical Corrigendum 1.* – Technical Corrigendum ITU-T Rec. X.680 (1997)/Tech. Corr. 1 (1999) | ISO/IEC 8824-1:1998/Cor.1:1999. http://www.itu.int/itudoc/itu-t/approved/x/x680cor1.html.

[ISO8824-1DTC2] *Information Technology - ASN.1: Specification of Basic Notation - Draft Technical Corrigendum 2.* – Draft Technical Corrigendum ITU-T Rec. X.680 (1997)/Draft Tech. Corr. 2 (2000) | ISO/IEC 8824-1:1998/DCor.2:2000. 71, 112, 139

[ISO8824-1DTC3] *Information Technology - ASN.1: Specification of Basic Notation - Draft Technical Corrigendum 3.* – Draft Technical Corrigendum ITU-T Rec. X.680 (1997)/Draft Tech. Corr. 3 (2000) | ISO/IEC 8824-1:1998/DCor.3:2000. 72, 143

[ISO8824-1DTC4] *Information Technology - ASN.1: Specification of Basic Notation - Draft Technical Corrigendum 4.* – Draft

Technical Corrigendum ITU-T Rec. X.680 (1997)/Draft Tech. Corr. 1 (2000) | ISO/IEC 8824-1:1998/DCor.4:2000. 72, 271, 273

[ISO8824-2] *Information Technology - Abstract Syntax Notation One (ASN.1): Information Object Specification.* – International Standard ITU-T Rec. X.681 (1997) | ISO/IEC 8824-2:1998. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x681.html. 302, 311, 332, 343, 355, 359, 513

[ISO8824-2Amd1] *Information Technology - ASN.1: Information Object Specification - Amendment 1: ASN.1 Semantic Model.* – Amendment ITU-T Rec. X.681 (1997)/Amd.1 (1999) | ISO/IEC 8824-2:1998/Amd.1:1999. http://www.itu.int/itudoc/itu-t/approved/x/x681amd1.html. 121

[ISO8824-2TC1] *Information Technology - ASN.1: Information Object Specification - Technical Corrigendum 1.* – Technical Corrigendum ITU-T Rec. X.681 (1997)/Tech. Corr. 1 (1999) | ISO/IEC 8824-2:1998/Cor.1:1999. http://www.itu.int/itudoc/itu-t/approved/x/x681cor1.html.

[ISO8824-3] *Information Technology - Abstract Syntax Notation One (ASN.1): Constraint Specification.* – International Standard ITU-T Rec. X.682 (1997) | ISO/IEC 8824-3:1998. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x682.html. 294, 350, 513

[ISO8824-3DTC1] *Information Technology - ASN.1: Specification of Basic Notation - Draft Technical Corrigendum 1.* – Draft Technical Corrigendum ITU-T Rec. X.682 (1997)/Draft Tech. Corr. 1 (2000) | ISO/IEC 8824-3:1998/DCor.1:2000. 296

[ISO8824-3DTC2] *Information Technology - ASN.1: Specification of Basic Notation - Draft Technical Corrigendum 2.* – Draft Technical Corrigendum ITU-T Rec. X.682 (1997)/Draft Tech. Corr. 2 (2000) | ISO/IEC 8824-3:1998/DCor.2:2000. 72, 283, 284

[ISO8824-4] *Information Technology - Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 Specifications.* – International Standard ITU-T Rec. X.683 (1997) | ISO/IEC 8824-4:1998. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x684.html. 379, 382, 384, 385, 386, 513

[ISO8824-4Amd1] *Information Technology - ASN.1: Parameterization of ASN.1 Specifications - Amendment 1: ASN.1 Semantic Model.* – Amendment ITU-T Rec. X.683 (1997)/Amd.1 (1999) | ISO/IEC 8824-4:1998/Amd.1:1999. http://www.itu.int/itudoc/itu-t/approved/x/x683amd1.html. 121

[ISO8825-1] *Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).* – International Standard ITU-T Rec. X.690 (1997) | ISO/IEC 8825-1:1998. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x690.html. 400, 406, 418, 513, 514

[ISO8825-1Amd1] *Information Technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) - Amendment 1: Relative Object Identifiers.* – Amendment ITU-T Rec. X.690 (1997)/Amd.1 (1999) | ISO/IEC 8825-1:1998/Amd.1:1999. http://www.itu.int/itudoc/itu-t/approved/x/x690amd1.html.

[ISO8825-1TC1] *Information Technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) - Technical Corrigendum 1.* – Technical Corrigendum ITU-T Rec. X.690 (1997)/Tech. Corr. 1 (1999) | ISO/IEC 8825-1:1998/Cor.1:1999. http://www.itu.int/itudoc/itu-t/approved/x/x690cor1.html.

[ISO8825-2] *Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER).* – International Standard ITU-T Rec. X.691 (1997) | ISO/IEC 8825-2:1998. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x691.html. 426, 428, 429, 448, 516

[ISO8825-2Amd1] *Information Technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER) - Amendment 1: Relative Object Identifiers.* – Amendment ITU-T Rec. X.691 (1997)/Amd.1 (1999) | ISO/IEC 8825-2:1998/Amd.1:1999. http://www.itu.int/itudoc/itu-t/approved/x/x691amd1.html.

[ISO8825-2TC1] *Information Technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER) - Technical Corrigendum 1.* – Technical Corrigendum ITU-T Rec. X.691 (1997)/Tech. Corr. 1 (1999) | ISO/IEC 8825-2:1998/Cor.1:1999. http://www.itu.int/itudoc/itu-t/approved/x/x691cor1.html.

[ISO8859] *Information Processing - 8-bit Single-Byte Coded Graphic Character Sets* - Part 1: Latin Alphabet No. 1, ISO 8859-1:1987. Part 2: Latin Alphabet No. 2, ISO 8859-2:1987. Part 3: Latin Alphabet No. 3, ISO 8859-3:1988. Part 4: Latin Alphabet No. 4, ISO 8859-4:1988. Part 5: Latin/Cyrillic Alphabet, ISO 8859-5:1988. Part 6: Latin/Arabic Alphabet, ISO 8859-6:1987. Part 7: Latin/Greek Alphabet, ISO 8859-7:1987. Part 8: Latin/Hebrew Alphabet, ISO 8859-8:1988. Part 9: Latin Alphabet No. 5, ISO 8859-9:1990. – International Standards ISO 8859:1990. http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=8859, http://park.kiev.ua/multiling/ml-docs/iso-8859.html, http://czyborra.com/charsets/iso8859.html. 172, 185

[ISO9066-2] *Open System Interconnection - Connection-mode* *protocol specifications - Reliable transfer: Protocol specification.* – International Standard ITU-T Rec. X.228 (1988) | ISO/IEC 9066-2:1989. http://www.itu.ch/itudoc/itu-t/rec/x/x200-499/x228.html. 25, 80, 516

[ISO9072-2] *Data networks and open system communication - Open System Interconnection - Service definitions - Remote operations: Protocol specification.* – International Standard ITU-T Rec. X.229 (1988) | ISO/IEC 9072-2:1989. http://www.itu.ch/itudoc/itu-t/rec/x/x200-499/x229.html. 25, 77, 154, 242, 280, 371, 374, 516

[ISO9596-1] *Information technology - Open Systems Interconnection - Common Management Information Protocol: Specification.* – International Standard ITU-T Rec. X.711 (1997) | ISO/IEC 9596-1:1998. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x711.html. 90, 159, 200, 244, 280, 514

[ISO9646-3] *Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN).* – International Standard ISO/IEC 9646-3:1998. http://www.iso.ch/cate/d30621.html. 478, 517, 521

[ISO9735] *Electronic data interchange for administration, commerce and transport (EDIFACT) – Application level syntax rules (Parts 1 to 9).* – Draft International Standard ISO/IEC 9735. http://www.iso.ch/cate/d28131.html, http://www.unece.org/trade/untdid/welcome.htm. 490, 514

[ISO9805-1] *Information technology - Open Systems Interconnection - Protocol for the commitment, concurrency and recovery service element: Protocol specification.* – International Standard ITU-T Rec. X.852 (93) | ISO/IEC 9805-1:1994. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x852.html. 26, 80, 513

[ISO9834-1] *Information Technology - OSI - Procedures for the operation of OSI Registration Authorities: General Procedures.* – International Standard ITU-T Rec. X.660 (1998) | ISO/IEC 9834-1:1998. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x660.html. 67, 68, 114, 154, 156, 157, 158, 162, 167, 198, 199, 404

[ISO9834-3] *Information Technology - OSI - Procedures for the operation of OSI Registration Authorities: Registration of values of RH-name-tree components for joint ISO and ITU-T use.* – International Standard ITU-T Rec. X.662 (1997) | ISO/IEC 9834-3:1997. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x662.html. 159

[ISO10165-4] *Information Technology - Open Systems Interconnection - Structure of management information - Part 4: Guidelines for the Definition of Managed Objects (+ Technical Corrigendum 1:1996 + Amendments 1, 3 and 4).* – International Standard ITU-T Rec. X.722 (1992) | ISO/IEC 10165-4:1992. http://www.iso.ch/cate/d18174.html. 90, 155, 159, 482, 484, 514

[ISO10646-1] *Information Technology - Universal Multiple-Octet Coded Character Set (UCS): Architecture and Basic Multilingual Plane.* – International Standard ISO/IEC 10646-1:1993. http://www.iso.ch/cate/d18741.html, ftp://ftp.unicode.org/Public/, http://www.unicode.org/, UNIDATA/UnicodeData-Latest.txt, http://www.dkuug.dk/JTC1/sc2/WG2/prot/, http://www.ifcss.org/ftp-pub/software/info/cjk-codes/Unicode.html (see also [Uni96]). 65, 172, 175, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 194, 196, 197, 273, 274, 306, 406, 442, 517, 537

[ISO10646-1Amd2] *Information Technology - Universal Multiple-Octet Coded Character Set (UCS): Architecture and Basic Multilingual Plane. Amendment 2: UCS Transformation Format 8 (UTF-8).* – International Standard ISO/IEC 10646-1:1993/Amd.2:1996. http://www.iso.ch/cate/d18741.html. 188, 190, 194, 406

[ISO11578] *Information Technology Open Systems Interconnection - Remote Procedure Call (RPC).* – International Standard ISO/IEC 11578:1996. http://www.iso.ch/cate/d2229.html. 488

[ISO13712-1] *Data networks and open system communication - Open System Interconnection - Service definitions - Remote operations: Concepts, model and notation (+ Technical Corrigendum 1:1995 + Amendment 1: Built-in operations).* – International Standard ITU-T Rec. X.880 (1994) | ISO/IEC 13712-1:1995. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x880.html, http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x880c1.html, http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x_880a1.html. 25, 80, 236, 248, 288, 295, 311, 375, 383

[ISO14750] *Information Technology - Open distributed processing - Interface Definition Language.* – International Standard ITU-T Rec. X.920 (1997) | ISO/IEC 14750:1998. 489

[ISOReg] *International Register of Coded Character Sets to be Used with Escape Sequences.* – Document ISO/IEC. http://www.itscj.ipsj.or.jp/ISO-IR/. 172, 175, 176, 178, 181, 182, 183, 193

[KDL87] Kong (M.), Dineen (T. H.), Leach (P. J.) et al. – *Network Computer System Reference Manual.* – Prentice-Hall, 1987. ISBN: 0-13-617085-4. 488

[Lar96] Larmouth (J.). – *Understanding OSI.* – International Thomson Computer Press, 1996. ISBN: 1-850321760, http://www.salford.ac.uk/iti/books/osi/all.html. 20, 57, 231, 242, 303, 492

[Lar99] Larmouth (J.). – *ASN.1 Complete.* – Morgan Kaufmann, 1999. ISBN: 0-12233-435-3, http://www.mkp.com/books_catalog/0-12233-435-3.asp, freely downloadable from http://www.oss.com/asn1/booksintro.html. xxiii

[LD97] Latu (G.) and Dubuisson (O.). – *Deux outils syntaxiques pour la notation ASN.1.* – Internal Report, France Télécom R&D, December 1997. Only in French. 469

[Lin93] Lin (H.-A. P.). – Estimation of the Optimal Performance of ASN.1/BER Transfer Syntax. *ACM SIGCOMM Computer Communication Review*, vol. 23, no. 3, 1993, pp. 45–58. 415

[MC93] Meyers (B. C.) and Chastek (G.). – *The Use of ASN.1 and XDR for Data Representation in Real-Time Distributed Systems.* – Technical Report CMU/SEI-93-TR-10, October 1993. http://www.sei.cmu.edu/publications/documents/93.reports/93.tr.010.html. 487

[Mit94] Mitra (N.). – An Introduction to the ASN.1 MACRO Replacement Notation. *AT&T Technical Journal*, May-June 1994, pp. 66–79. 312

[MMS79] Mitchell (J. G.), Maybury (W.) and Sweet (R.). – *Mesa Language Manual.* – Technical Report CSL-79-3 (Version 5.0), April 1979. 60

[OFM96] Olsen (A.), Færgemand (O.), Møller-Pedersen (B.), Reed (R.) and Smith (J.). – *Systems Engineering Using SDL-92.* – North-Holland, Elsevier, 1996, 3rd edition. ISBN: 0-444-89872-7, http://www.elsevier.nl/inca/publications/store/5/2/4/7/0/7/. 475, 476

[OPR96] Otte (R.), Patrick (P.) and Roy (M.). – *Understanding CORBA.* – Prentice Hall PTR, 1996. ISBN: 0-13-459884-9, http://www.prenhall.com/allbooks/ptr_0134598849.html. 488

[PC93] Piscitello (D. M.) and Chapin (A. L.). – *Open Systems Networking, TCP/IP and OSI.* – Addison-Wesley Professional Computing Series, 1993. ISBN: 0-201-56334-7, http://cseng.aw.com/bookdetail.qry?ISBN=0-201-56334-7&ptype=1182. 20, 156

[PR89] Partridge (C.) and Rose (M. T.). – A Comparison of External Data Formats. pp. 233–245. – Elsevier Science Publishers B.V. 488

[Ram98] Raman (L.). – OSI Systems and Network Management. *IEEE Communications Magazine*, March 1998, pp. 46–53. 480

[RFC822] Crocker (David H.). – *Standard for the Format of ARPA Internet Text Messages*. – Request For Comments 822, August 1982. http://www.faqs.org/rfcs/rfc822.html (see also [RFC2156]). 490

[RFC1213] McCloghrie (K.) and Rose (M.). – *Management Information Base for Network Management of TCP/IP-based Internets*. – Request For Comments 1213, March 1991. http://www.faqs.org/rfcs/rfc1213.html. 151

[RFC1832] Srinivasan (R.). – *XDR: External Data Representation Standard*. – Request For Comments 1832, August 1995. http://www.faqs.org/rfcs/rfc1832.html. 486

[RFC2234] Crocker (D.) and Overell (P.). – *Augmented BNF for Syntax Specifications: ABNF*. – Technical Report 2234, November 1997. http://www.faqs.org/rfcs/rfc2234.html. 490

[RFC2156] Kille (S.). – *MIXER (Mime Internet X.400 Enhanced Relay): Mapping between X.400 and RFC 822/MIME*. – Technical Report 2156, January 1998. http://www.faqs.org/rfcs/rfc2156.html. 176, 535

[RFC2279] Yergeau (F.). – *UTF-8, a transformation format of ISO 10646*. – Request For Comments 2279, January 1998. http://www.faqs.org/rfcs/rfc2279.html. 191, 406

[Rin95] Rinderknecht (C.). – *Parsing ASN.1:1990 with Caml Light*. – Internal Report 171, INRIA Sophia Antipolis, 1995. ftp://ftp.inria.fr/INRIA/publication/RT/RT-0171.ps.gz, http://pauillac.inria.fr/~rinderkn. 235, 374, 468

[RSA93] RSA Laboratories. – *PKCS #7: Cryptographic Message Syntax Standard.* – Technical Report, November 1993. 87

[Sch94] Schröder (R.). – *SDL'92 Data Handling in Combination With ASN.1.* – Master Thesis, Humboldt-Universität zu Berlin, March 1994, 97 p. 475

[SN93] Sample (M.) and Neufeld (G.). – Implementing Efficient Encoders and Decoders For Network Data Representations. *In: IEEE INFOCOM, Conference on Computer Communications, San Francisco*, ed. by IEEE Computer Society Press, pp. 1144–1153. – http://www.eecs.harvard.edu/cgi-bin/selectrefs.pl?KEYS=SAMPLE93. 487, 488

[Ste93] Steedman (D.). – *ASN.1 The Tutorial & Reference.* – Technology Appraisals Ltd., 1993. ISBN: 1-871802-06-7, http://www.techapps.co.uk/asn1gloss.html. 60, 235, 244, 373

[T.101] *Terminals for telematic services.* – Recommendation ITU-T Rec. T.101 (1994). http://www.itu.ch/itudoc/itu-t/rec/t/t101.html. 180

[T.433] *Terminals for Telematic Services - Document Transfer And Manipulation (DTAM) - Services and protocols - Protocol specification (+ Amendment 1, 1995: Revisions of T.433 to support G4 colour and file transfer).* – Recommendation ITU-T Rec. T.433 (1992). http://www.itu.ch/itudoc/itu-t/rec/t/t433.html, http://www.itu.ch/itudoc/itu-t/rec/t/t433amd1.html. 23, 514

[Tan96] Tanenbaum (A.). – *Computer Networks.* – Prentice-Hall, 1996. ISBN: 0-13-349945-6, http://www.prenhall.com/divisions/ptr/tanenbaum/. 8, 9, 20, 27

[TMF96] TeleManagement Forum, X/Open. – *ASN.1/C++. Application Programming Interface.* – Report 1.0 (draft 10a), 1996. ftp://ftp.tmforum.org/nmfsets/component/cs322/

NMF040-1.doc, ftp://ftp.tmforum.org/nmfsets/component/cs322/NMF040-2.doc. 142, 294, 466, 469, 489

[TH] Taylor (G.) and Howard (A.). – *A Study to Draw a Correlation Between EDIFACT and ASN.1.* – Technical Report. Department of Trade and Industry, Information Technology Standards Unit, Londres. 492

[Uni96] Unicode Consortium. – *The Unicode Standard, Version 2.0.* – Addison-Wesley, 1996. ISBN: 0-201-48345-9, with CD-ROM, http://cseng.aw.com/bookdetail.qry?ISBN=0-201-48345-9&ptype=0 (see also [ISO10646-1]). 65, 183, 188, 189, 191, 194, 196, 517, 532

[W3C00] *XML Schema.* – W3C Standard, April 4, 2000. http://www.w3.org/TR/xmlschema-0/, http://www.w3.org/TR/xmlschema-1/, http://www.w3.org/TR/xmlschema-2/. See also http://www.w3schools.com/schema/. 271, 456

[WBS90] Wu (W.), Bilgic (M.) and Sarikaya (B.). – VHDL Modeling and Synthesis of an ASN.1 Encoder/Decoder. *In: CCVLSI Conference, Ottawa*, pp. 1.5.1–1.5.5. 471

[Whi89] White (J.). – ASN.1 and ROS: The Impact of X.400 on OSI. *IEEE Journal on Selected Areas in Communications*, vol. 7, no. 7, September 1989. 60

[Wil99] Willcock (C.). – New Directions in ASN.1: Towards a Formal Notation for Transfer Syntax. *In: IWTCS Conference. Testing of Communicating Systems. Methods and Applications*, ed. by Csopaki (G.), Dibuz (S.) and Tarnay (K.), ed. by Publishers (K. A.), pp. 31–40. 457

[X.121] *Data networks and open system communication - Public data networks - Network aspects: International numbering plan for public data networks.* – Recommendation ITU-T Rec. X.121 (1996). http://www.itu.ch/itudoc/itu-t/rec/x/x1-199/x121.html. 158

[X.400] *Non-telephone telecommunication services Message handling services - Message handling system and service overview.* – International Standard ITU-T Rec. F.400/X.400 (1996) | ISO/IEC 10021-1:1997. http://www.itu.ch/itudoc/itu-t/rec/x/x200-499/x400.html. 26, 155, 157, 159, 231, 515

[X.409] *Message Handling Systems: Presentation Transfer Syntax and Notation.* – Recommendation ITU-T Rec. X.409 (1984). 151, 174, 176, 210, 211, 226, 241, 364, 394

[X.435] *Data networks and open system communication -Message Handling Systems: Electronic data interchange messaging system (+ Technical corrigendum 1, 1998 + Amendment 1, 1997: Compression extension).* – International Standard ITU-T Rec. X.435 (1991) | ISO/IEC 10021-9:1995. http://www.itu.ch/itudoc/itu-t/rec/x/x200-499/x435.html, http://www.itu.ch/itudoc/itu-t/rec/x/x200-499/x435cor1.html, http://www.itu.ch/itudoc/itu-t/rec/x/x200-499/x435am1.html. 155, 162

[X.501] *Information technology - Open Systems Interconnection - The directory: Models.* – International Standard ITU-T Rec. X.501 (1997) | ISO/IEC 9594-2:1997. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x501.html. 243, 329, 345, 346

[X.509] *Information technology - Open Systems Interconnection - The directory: Authentication framework.* – International Standard ITU-T Rec. X.509 (1997) | ISO/IEC 9594-8:1997. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x509.html, http://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.htm. 89, 145, 243, 375, 420, 422

[X.519] *Information technology - Open Systems Interconnection - The directory: Protocol specifications.* – International Standard ITU-T Rec. X.519 (1997) | ISO/IEC 9594-5:1997. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x519.html. 288

[X.520] *Information technology - Open Systems Interconnection - The directory: Selected attribute types.* – International

Standard ITU-T Rec. X.520 (1997) | ISO/IEC 9594-6:1997. http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x520.html. 237, 342, 379

[Xer81] XEROX CORPORATION. – *Courier*. – Xerox System Integration Bulletin OPD B018112, 1981. 60

[Z.100] *Programming languages - Formal description techniques (FDT) - Specification and Description Language (SDL)*. – Pre-published Recommendation ITU-T Rec. Z.100 (1999). http://www.itu.ch/itudoc/itu-t/approved/z/z100.html. 474, 517, 539

[Z.100S1] *SDL+ methodology: Use of MSC and SDL (with ASN.1)*. – Supplement 1 to Recommendation [Z.100], ITU-T Rec. Z.100 Suppl. 1 (1997). http://www.itu.ch/itudoc/itu-t/rec/z/z100sup1.html. 475

[Z.105] *Programming languages - Formal description techniques (FDT) - Specification and Description Language (SDL): SDL combined with ASN.1 modules*. – Pre-published Recommendation ITU-T Rec. Z.105 (1999). http://www.itu.ch/itudoc/itu-t/approved/z/z105.html. 475, 476, 478, 507, 508, 509, 510, 511

[Z.107] *Programming languages - Formal description techniques (FDT) - Specification and Description Language (SDL): SDL with embedded ASN.1*. – Recommendation ITU-T Rec. Z.107 (1999). http://www.itu.ch/itudoc/itu-t/rec/z/z-107.html. 475, 478

[Z.140] *Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3 – TTCN-3: Core Language*. – Draft Revised ITU-T Recommendation X.292/Z.140. 480

Index

The entries in *italics* stand for ASN.1 grammar productions (e.g. *ActualParameter*); page numbers in brackets indicate that the corresponding grammar production is only used on that page but not defined there. The lexical tokens (e.g. bstring) are written in sans serif font. The ASN.1 keywords (e.g. ABSTRACT-SYNTAX), the symbols (e.g. "!") and some identifiers used in the ASN.1 standard (e.g. administration) are highlighted in teletype font.