ASN.1-C COMPILER FOR AUTOMATIC PROTOCOL IMPLEMENTATION

By

YUELI YANG

B.Eng., Tsinghua University, China, 1985

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 1988

© Yueli Yang, 1988
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science
The University of British Columbia
Vancouver, Canada

Date October 15, 1988
Abstract

One of the basic requirements of communication protocols in a heterogeneous computer network is a standard external data-transfer representation. Abstract Syntax Notation One (ASN.1) has been widely used in international standard specifications. Its transfer-syntax of Basic Encoding Rules (BER) is applied as the standard external data representation. This thesis presents an efficient BER implementation, called the ED library. The ED library includes a number of encoding and decoding routines that may be used as primitive functions to compose encoders and decoders for arbitrarily complicated ASN.1 data-types. The Performance of the ED library is measured and discussed.

Based on the ED library, an ASN.1-C compiler, called CASN1, is designed and implemented to release communication software programmers from the arduous work of translating protocol-defined data-types and constructing their encoders and decoders. Given an ASN.1 protocol specification, CASN1 automatically translates the input ASN.1 modules into C and generates the BER encoders and decoders for the protocol defined data-types. This thesis discusses the design principles, user interface, internal structures, and the implementation and of CASN1. Example applications are given. Both the ED library and CASN1 are implemented in C on UNIX 4.2 BSD using the YACC and LEX tools.
# Contents

Abstract .................................................. ii

Contents ................................................. iii

List of Tables ........................................... vi

List of Figures ........................................... vii

Acknowledgement ......................................... ix

1 Introduction ........................................... 1
   1.1 Motivation ......................................... 1
   1.2 An Example of using ASN.1 and BER ................. 4
   1.3 Related Work ....................................... 8
       1.3.1 BBN Message Transfer Protocols Project ........ 9
       1.3.2 The ISODE Data Service ......................... 11
       1.3.3 Ean Encoder and Decoder ......................... 14
       1.3.4 Thesis Statements ............................... 14
   1.4 Thesis Organization ................................. 16

2 The OSI ASN.1 and BER ................................ 17
   2.1 Abstract Syntax Notation One (ASN.1) ............... 17
   2.2 Basic Encoding Rules (BER) ......................... 21
   2.3 Differences between X.409 and ISO ASN.1/BER ........ 23

3 Data Type Translation between ASN.1 and C .......... 24
   3.1 Primitive Types ................................... 24
   3.2 Bit String and Octet String Types ................ 26
   3.3 Object Identifier Type ............................. 27
   3.4 Any Type .......................................... 27
   3.5 Sequence and Set Types ............................ 27
   3.6 Sequence of and Set of Types ...................... 28
4 The ED Library Design and Implementation

4.1 The ED Library Design

4.1.1 The ED Library Utility Routines
4.1.2 Primitive Type Encoding and Decoding
4.1.3 Structured Type Encoding and Decoding
4.1.4 Tagged Type Encoding and Decoding
4.1.5 Special Feature of Decoding
4.1.6 The ANY Type Encoding and Decoding
4.1.7 The REAL Type Encoding and Decoding

4.2 The ED Library Implementation

4.2.1 The ED Library Data Structures
4.2.2 The ED Library Routines
4.2.3 The ED Library Sub-memory System

4.3 Using the ED Library Routines

5 The CASN1 Design and Implementation

5.1 The CASN1 Design

5.1.1 The Interfaces of CASN1
5.1.2 The Functions of CASN1
5.1.3 Modification to ASN.1 Syntax

5.2 The CASN1 Implementation

5.2.1 The First pass
5.2.2 The Second pass
5.2.3 The Third pass

5.3 Using the CASN1 Output

6 Performance and Applications

6.1 The ED Library Performance
6.2 Results of CASN1 Applications

6.2.1 The ISO Virtual Terminal Protocol
6.2.2 The CCITT X.266 Presentation Protocol
6.2.3 The OSI X.500 Directory Protocol
6.2.4 The ASN.1/Estelle Integration Project

7 Conclusions and Future Research

7.1 Conclusions
7.2 Future Work
Bibliography

A BNF of ASN.1 Syntax Rules 82
B BNF of CASN1 Syntax Rules 90
C Example of Structured Type Encoding and Decoding 100
D Example of CHOICE Type Encoding and Decoding 106
E Example of Using CASN1 and ED Library 112
List of Tables

3.1 Data-Type Translation between ASN.1 and C ................................. 25
4.1 The ED Library Primitive Class Routines .................................... 36
6.1 The ED Library Performance ..................................................... 66
6.2 OCTET STRING Encode Octet Complexity ..................................... 67
6.3 BASICVTITEM Encode Octet Complexity .................................... 67
6.4 PERSONNELRECORD Encode Octet Complexity ............................. 67
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Communicate Using an External Data Representation</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>write.c</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>read.c</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>ed_write.c</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>ed_read.c</td>
<td>6</td>
</tr>
<tr>
<td>1.6</td>
<td>Execution Script of the Example</td>
<td>7</td>
</tr>
<tr>
<td>1.7</td>
<td>BBN X.409 Compiler and Parser</td>
<td>10</td>
</tr>
<tr>
<td>1.8</td>
<td>BBN Data Encoding and Decoding</td>
<td>11</td>
</tr>
<tr>
<td>1.9</td>
<td>ISODE ASN.1 Compiler</td>
<td>13</td>
</tr>
<tr>
<td>1.10</td>
<td>Interface of Encoder and Decoder</td>
<td>15</td>
</tr>
<tr>
<td>2.1</td>
<td>An Example of Data Encoding</td>
<td>21</td>
</tr>
<tr>
<td>2.2</td>
<td>Structure of BER Transfer Syntax</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>C Structure BITS and OCTS Definition</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>C Structure 0ID Definition</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>C Structure LIST Definition</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>C Structure TIME Definition</td>
<td>30</td>
</tr>
<tr>
<td>3.5</td>
<td>C Structure EXT Definition</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>Function of Encoding and Decoding Routine</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>C Structure IOP Definition</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>C Structure IDX Definition</td>
<td>41</td>
</tr>
<tr>
<td>4.4</td>
<td>The ED Library Stack Definition</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>The Interfaces of CASN1</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>BNF of OPERATION Definition and ERROR Definition</td>
<td>54</td>
</tr>
<tr>
<td>5.3</td>
<td>Using CASN1 Output</td>
<td>63</td>
</tr>
<tr>
<td>6.1</td>
<td>OCTET STRING Encode Octet Complexity</td>
<td>68</td>
</tr>
<tr>
<td>6.2</td>
<td>BASICVTPITEM Encode Octet Complexity</td>
<td>69</td>
</tr>
<tr>
<td>6.3</td>
<td>PERSONNELRECORD Encode Octet Complexity</td>
<td>70</td>
</tr>
<tr>
<td>6.4</td>
<td>OCTET STRING Encode/Decode Time Complexity</td>
<td>72</td>
</tr>
</tbody>
</table>
6.5 PERSONNEL RECORD Encode/Decode Time Complexity ....... 73
6.6 BASICVTITEM Encode/Decode Time Complexity .......... 74
Acknowledgement

Some moments in our lives seem eternal, whether in time, or in depth. The years I spent in Vancouver were full of such moments. It is not within the following limited lines that I will be able to do justice to all of them.

First of all, I would like to sincerely thank my supervisor Dr. Gerald W. Neufeld for many things, especially for suggesting the thesis topic, for fruitful discussions and wonderful guidance, and for being a continuous source of advice and encouragement during the course of this research and my stay in Vancouver.

This thesis would not be what it is without the help from many friends. Thanks to B. Hilpert and E. Lau for their help in discussing and debugging CASN1; to D. Ouimet, M. Goldberg and S. Ng for providing comments by using this package in their theses and research; and to I. Chan and J. Demco for their help in collecting information about the related works.

Thanks to Dr. Samuel T. Chanson for being the second reader of this thesis and to L. Liang, B. Brachman, B. Hilpert, M. Goldberg, and H. Kreyszig for reading, commenting and correcting grammar errors in many drafts of this thesis.

Thanks to the Department of Computer Science at the University of British Columbia for providing the computing equipment to facilitate this research and for providing financial support in the form of teaching and research assistantships during my M.Sc graduate studies.

And last, special thanks should go to my dear husband Luping and my parents. Without their consistent support and invaluable love, I would never be able to conquer the obstacles that seemed insurmountable.
Chapter 1

Introduction

As the introduction to the thesis, this chapter presents the motivation of the thesis research, discusses the related work, and specifies the thesis statements. An outline of the thesis organization is given in the last section of this chapter.

1.1 Motivation

In a heterogeneous computer communication network, computers may differ in their internal data representations. For example, characters may be coded in ASCII, EBCDIC, or in a graphical character set; integers may vary in length from 8 to 64 bits; bytes may be addressed from left to right or from right to left within a word. Also, programming languages may differ in their representation of numbers or data structures. Therefore, when transferring data between different types of computers or various application entities implemented in different programming languages, the specification of the data structure has to be transferred as well.

In an open system environment, it is impractical to provide a separate data representation transformation mechanism for each possible pair of computer types, as this would lead to \( N \times (N-1) \) translators for \( N \) different types of computers, without considering language differences. As shown in Figure 1.1, a practical scheme may be to define a network-wide external data
CHAPTER 1. INTRODUCTION

representation for data transfer\textsuperscript{1}. Senders encode message data from local representations into the external representation before sending messages out to the network. Receivers decode the received message data from the external representation into their local representations before processing the messages. With such an external data representation, only 2\(N\) translators (one to and one from the local representation for each of the computers or languages) are needed.

![Diagram](https://via.placeholder.com/150)

Figure 1.1: Communicate Using an External Data Representation

The purpose of the presentation-layer in the OSI model\textsuperscript{2} is to overcome the differences of data representation in heterogeneous computer systems. It functions between the local data representations and the external data representations. The presentation-layer provides facilities to represent application-layer data in a way that is independent of both computers and application languages. It also provides protocols to enhance the session-layer service by providing external representations for data transfer, as well as mechanisms to define and to select these representations.

\textsuperscript{1}This figure is from [19].

\textsuperscript{2}The OSI seven-layer model (Open System Interconnection reference model) is defined by ISO (the International Standards Organization). Interested readers may refer to "Computer Networks", the second edition by A. S. Tanenbaum, Prentice-Hall, 1988.
CHAPTER 1. INTRODUCTION

The languages defined in the Courier protocol developed at Xerox and in the eXternal Data Representation (XDR) developed by Sun Microsystems, Inc. are examples of such external data representations. The Abstract Syntax Notation One (ASN.1), defined by ISO, has been used in many international standards to specify communication protocols. The Basic Encode Rules (BER), also defined by ISO, are used to encode data of ASN.1 types into a transfer syntax (octet sequence) which is also an external data representation.

ASN.1 is an abstract syntax notation (or a language) for defining both complex types and the values of these types without worrying about how the instances of these types are represented during data transfer. It can be used in defining the abstract-syntax of information, particularly in specifying standards for application protocols. The BER is used to transform the data-values of ASN.1 types into an octet-sequence for data transfer. Originally, ASN.1 and BER were defined in CCITT recommendation X.409 [4] in 1984. In 1985, the ISO 8824 standard (draft) [7] first defined the ASN.1 syntax independent of its encoding rules. Meanwhile, the ISO 8825 standard (draft) [9] defined BER as one of the ASN.1 encoding/decoding rules. Both ISO 8824 and ISO 8825 are technically aligned with the relevant parts of CCITT X.409. ASN.1 and BER have been refined several times in 1985, 1986, and 1987 and their final versions were published in December 1987. At the same time, their first addendum — extensions to ASN.1 [8] and to ASN.1 Basic Encoding Rules [10] were also published.

Abstract-syntax is a formal syntax notation which specifies information independent of the information representation. Transfer-syntax is a syntax notation which represents information in octet sequences. In this thesis, the terminology transfer-syntax, external data-representation, and octet-sequence have the same meaning and are used interchangeably.
1.2 An Example of using ASN.1 and BER

As a formal notation, ASN.1 can be applied whenever it is necessary to define information in an abstract-syntax. It is particularly applicable to application protocol specifications that have complex data types. BER translate the data-values of ASN.1 types into a transfer-syntax before transmitting the data over the network. The following is an example of using BER to transfer data of ASN.1 INTEGER type in a heterogeneous environment.

In the Department of Computer Science at University of British Columbia, there are three different types of machines connected by an Ethernet: machine *grads* is a SUN 3/260, *ugly* is a SUN 2/50 and *ean* is a VAX 11/750. They all run the version of UNIX 4.2 BSD\(^4\) operating system. Since the SUN and VAX adopt different data representations internally, the three machines on the Ethernet constitute a heterogeneous environment at the machine-level.

The executable code of *write.c* and *read.c* are stored in the files *write* and *read* respectively. *Read* reads ten integers 0,1,2, ..., 9 from *stdin* and *write* displays them on *stdout*. The files *ed_read* and *ed_write* contain the executable code of *ed_write.c* and *ed_read.c* respectively. They are almost identical to *read.c* and *write.c*, except they transform the data into ASN.1 transfer-syntax during data transfer. *Encode_int()* and *Decode_int()* in the files *ed_write.c* and *ed_read.c* are the encoder and decoder\(^5\) of ASN.1 INTEGER type. The four programs are shown in Figures 1.2, 1.3, 1.4, and 1.5 respectively, and are compiled and installed on each of the three machines.

\(^4\)UNIX is a trade mark of Bell Laboratories and 4.2 BSD is a version of the UNIX operating system distributed by the University of California, at Berkeley.

\(^5\)Encode transforms a data-value from local representation into the ASN.1 octet-sequence according to BER. *Decode* refers to the inverse of encode. The terms *encode routine*, *encoding routine* and *encoder* are used interchangeably in this thesis to refer to the encode function. Similarly, *decode routine*, *decoding routine* and *decoder* are used interchangeably to refer to the decode function.
main()  /* write.c */
{
    long i;
    for (i = 0; i < 8; i++) {
        if (fwrite((char *) &i, sizeof(i), 1, stdout) != 1) {
            fprintf(stderr, "failed! \n");
            exit(1);
        }
    }
}

Figure 1.2: write.c

main()  /* read.c */
{
    long i, j;
    for (j = 0; j < 8; j++) {
        if (fread((char *) &i, sizeof(i), 1, stdin) != 1) {
            fprintf(stderr, "failed! \n");
            exit(1);
        }
        printf("%x ", i);
    }
    printf("\n");
}

Figure 1.3: read.c
Figure 1.4: ed_write.c

```c
main() /* ed_write.c */
{
    long i, len;
    byte *b;

    for (i = 0; i < 8; i++) {
        b = serIDX (Encode_int(UNIVERSAL, PRIMITIVE, CODE_INTEGER, &i),
                    &b, &len);
        if (fwrite((char *)b, sizeof(*b), len, stdout) != len) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
    }
}
```

Figure 1.5: ed_read.c

```c
main() /* ed_read.c */
{
    long i, j;
    byte b[10], *c;

    for (j = 0; j < 8; j++) {
        if (fread((char *)b, sizeof(byte), 3, stdin) != 3) {
            printf(stderr, "failed!\n");
            exit(1);
        }
        c = b;
        Decode_int(UNIVERSAL, PRIMITIVE, CODE_INTEGER, &c, &i);
        printf("%x ", i);
    }
    printf("\n");
}
```
Figure 1.6: Execution Script of the Example

(1) rsh grads write | rsh grads read
0 1 2 3 4 5 6 7

(2) rsh grads write | rsh ugly read
0 1 2 3 4 5 6 7

(3) rsh grads write | rsh ean read
0 1000000 2000000 3000000 4000000 5000000 6000000 7000000

(4) rsh grads ed_write | rsh ean ed_read
0 1 2 3 4 5 6 7

(5) rsh grads ed_write | rsh ugly ed_read
0 1 2 3 4 5 6 7

(6) rsh grads ed_write | rsh grads ed_read
0 1 2 3 4 5 6 7
CHAPTER 1. INTRODUCTION

Figure 1.6 is the execution script. Executions (1) and (2) show the correct data transfer between machines of the same type (two SUN's) without using an external data representation. Execution (3) shows the transfer error when passing data between two different machines (VAX and SUN) without using an external data representation. Execution (4) shows that the problem is overcome with ASN.1 as the external data representation. Executions (5) and (6) demonstrate that ed_write() and ed_read() can also be used when communicating with the same kind of machine, although it may not always be necessary.

The above example also demonstrates the steps of using ASN.1 and BER in data transfer:

1. **Define data in ASN.1 syntax.** According to ASN.1, integer X is defined as: \( X ::= \text{INTEGER} \).

2. **Translate the type-definition from ASN.1 into an application programming language.** In our example, the application programming language is C. So, the type-definition is translated to: \( \text{typedef int } X \).

3. **Write the encoder and decoder for the data type according to BER.** In our example, the encoder is Encode_int() and the decoder is Decode_int().

4. **Write communication programs to manage data transfer over the communication network.**

### 1.3 Related Work

As a formal abstract-syntax notation, ASN.1 has been widely used in many international standards. To implement communication protocols in a heterogeneous computer network environment, one of the basic requirements is data encoding and decoding; i.e., converting data-values between the local representations and the external transfer representation. As a result, a considerable amount of work has been done in implementing BER and ASN.1 compilers. This
section briefly discusses some related work. Readers are assumed to have some knowledge of
ASN.1 to appreciate this section. Naive readers can refer to Chapter 2 for a summary of ASN.1
and BER. For detailed information on the related work, interested readers should refer to the
relevant documents cited in the bibliography.

1.3.1 BBN Message Transfer Protocols Project

In 1984, A. R. Pope and D. P. Deutsch at Bolt Beranek and Newman Inc. implemented
an X.409 presentation-layer data-transfer system in C on UNIX [14]. It is part of a project on
message transfer protocols for the National Bureau of Standards (NBS).

Their X.409 presentation-layer data-transfer system is a collection of tools that process
type-notation, value-notation, and data encoding/decoding. In particular, these tools en­
code/decode PDUs (Protocol Data Units) between the local representation and the external
standard transfer-syntax representation.

The tools include a compiler which compiles X.409 modules of type-notations into data
structures, called type tables. A type table is essentially a tree. Its hierarchy corresponds
directly to the manner in which constructor types⁶ are composed. The tree nodes correspond
to individual primitive⁷ or constructor types and tree edges denote the composition of complex
constructor types from simpler ones. The type tables are used to process the value-notations
and data encoding/decoding.

The tools also include a parser which parses X.409 value-notations and translates them into
data structures, called value tables. Similar to the type tables, each value table is also
organized as a tree, corresponding to the structure of the value's type. The tree nodes represent

⁶ Constructor types which is the same as structured types are defined in Section 2.1
⁷ Primitive types are also defined in Section 2.1
the values of primitive or constructor types and tree edges denote the data-value composition. The \textit{value tables} are used in data encoding and decoding.

Both the \textit{type table} and the \textit{value table} are C data structures. They are stored in a file which may be included by user programs. Figure 1.7 shows the process of compiling X.409 type-notations and parsing X.409 value-notations.

Two functions are passed to the parser as parameters:

1. The \textit{type table} describes the data structure of the input value-notation and directs the parsing.

2. The \textit{lexical} does the lexical analysis of the value-notation and provides tokens to the parser.

A presentation-layer data value represented by a \textit{value table} can be encoded by the routine \texttt{encode_value()} to produce a BER octet-sequence. A presentation-layer data-value in
its encoded form (i.e., BER octet-sequence) can be decoded by the routine `decode_value()` to produce a value table. The type table of the value is passed to `encode_value()` and `decode_value()` as one of the parameters to guide the encoding/decoding process. Figure 1.8 shows the process of data encoding and decoding.

![Figure 1.8: BBN Data Encoding and Decoding](image)

1.3.2 The ISODE Data Service

The ISODE (ISO Development Environment) is a system developed by the Wollongong Group and the Northrop Co. [13]. It is an implementation of some ISO protocols in C on UNIX as well as on several other systems. The software includes implementations of the OSI Association Control Service, Remote Operation Service, Reliable Transfer Service, and abstract-syntax and transfer mechanisms. It also includes the implementation of the OSI presentation-layer service, session-layer service, and transport-layer service. Some application programs and user interfaces are also implemented using their software.
The *libpsap* library implements presentation-layer abstract-syntax for the machine independent exchange of data structures. It manipulates two objects: *presentation-elements PE* and *presentation-streams PS*. PE is used to represent an arbitrarily complex data structure in a machine-independent form. Several routines have been implemented to translate between the machine-independent representation PE and machine-specific objects, such as integers, strings, and the like. A PS is an object to represent an I/O path of a PE data structure, such as a communication port or a file pointer.

The three programs *rosy*, *posy* and *pepy* together function as an ASN.1 compiler. Figure 1.9 shows the organization of the ISODE ASN.1 compiler.

The *remote operation module* is an ASN.1 module-definition with extensions to remote operation specifications. *Rosy* reads the description of a *remote operation module* and produces the corresponding C-stubs and C-definitions which are used in accessing the remote operation services. It also produces an *abstract-syntax module* which is simply a copy of the type-definitions in the *remote operation module*. *Posy* reads the description of an *abstract-syntax module* (possibly produced by *rosy*) and produces the corresponding C structure-definitions and an *augmented abstract-syntax module*. The C structure-definitions are used by the remote operation invoker and performer. The run-time environment is responsible for mapping these data structures to the abstract-syntax stored in PE and then mapping the abstract-syntax from PE to the transfer-syntax. *Pepy* reads the description of an *augmented abstract-syntax module* (probably produced by *posy*) and produces three C programs — PE parser, PE constructor and PE printer — to parse, create, and print presentation-elements (PE) of the objects described in the module. The PE constructor maps a data structure into abstract-syntax in PE and the PE parser does the inverse — mapping a presentation-element in PE into a C data structure. The syntax of the *augmented*
Figure 1.9: ISODE ASN.1 Compiler
abstract-syntax module is based on ASN.1 with several extensions: compiler directives, action statements, control statements and value passing statements. The ISODE ASN.1 compiler does not recognize ASN.1 macro-definitions. It ignores the ASN.1 type-definition of COMPONENTS OF, SELECTION and DEFAULT types.

1.3.3 Ean Encoder and Decoder

The Ean messaging system developed by a research group in the Computer Science Department at the University of British Columbia contains another implementation of BER. It has a number of encoding and decoding procedures to transform data between a C structure called an E_NODE and BER transfer-syntax. An E_NODE has five fields. One field records the tag of an ASN.1 type. One field records the data length. The other three fields are pointers to the primitive data value, to another E_NODE if the data is of constructor type and to another E_NODE if the data has sibling nodes (other values on the same level). The E_NODE can be viewed as a tree structure where each node corresponds to a primitive or a constructor value. The edges (represented by the E_NODE pointer fields) indicate the relationship between the types defined in tree nodes. With the E_NODE structure, a data-value can be translated into the BER transfer-syntax by calling the corresponding encoding procedure. The data-value in transfer-syntax may also be decoded back into an E_NODE by calling the corresponding decoding procedure. The E_NODE has been implemented as an abstract data type.

1.3.4 Thesis Statements

The three BER implementations discussed in this chapter use the same idea: An internal data structure is used to represent data in an abstract-syntax. The internal data structure is also used as the interface to the encoding and decoding routines. This is shown in Figure 1.10.
CHAPTER 1. INTRODUCTION

Figure 1.10: Interface of Encoder and Decoder

The type table is the internal data structure in the BBN implementation, as well as the PE_Type in the ISODE implementation and the E_NODE in the Ean implementation. With such an internal data structure as the interface, some overhead must be paid for converting data between user data and this internal data structure during encoding and decoding. This thesis presents another BER implementation — ED library — which avoids this overhead.

The ED library includes a number of BER encoding and decoding routines for ASN.1 defined types. They are implemented in C on UNIX 4.2 BSD. The encoding routines transform data directly from C to BER transfer-syntax. The decoding routines do the inverse, transforming data from BER transfer-syntax to C. If the transfer-syntax is from a file and is OCTET STRING or BIT STRING type in CONSTRUCTOR form (usually it contains a large data-value), the decoding routine provides a special function to decode only the data structure, and the data-value can be loaded later from the file. The ED library is implemented based on the ISO 8825 BER-definition and its addendum. The CCITT Recommendation X.208 and X.209 [1,2] are also referenced. Since data encoding and decoding are frequently used during data transformation, efficiency was the top concern in the design and implementation of the ED library.

Based on the ED library, this thesis also presents a design and implementation of an ASN.1-
CHAPTER 1. INTRODUCTION

C compiler, named CASN1. It is a package written in C on UNIX 4.2 BSD using the YACC and LEX tools. It parses an ASN.1 module-definition and translates all type-definitions into C. It also generates BER encoding/decoding routines for all defined ASN.1 types. CASN1 supports all ASN.1 types, including COMPONENTS OF, SELECTION type and ENCRYPTED type. It ignores the ASN.1 macro-definitions. However, it has the extensions for OPERATION-definitions and ERROR-definitions in its syntax for specifying abstract operations. The operation arguments, results and error parameters are recognized as ASN.1 type-definitions by CASN1. Their encoding and decoding routines are generated and stored in files with the operation number or error number as indices. CASN1 is implemented based on the ISO 8824 ASN.1-definition and its addendum. The CCITT Recommendation X.208 is also referenced.

1.4 Thesis Organization

Chapter 2 briefly introduces the international standards for ASN.1 and BER. Knowledgeable readers can skip this chapter. Chapter 3 discusses the translation between ASN.1 and C. Chapter 4 presents the design and implementation of the ED library. Chapter 5 explains the design and implementation of CASN1. Chapter 6 shows the performance and some application examples of the ED library and CASN1. Chapter 7 summarizes the thesis and points out some future work. Appendix A and Appendix B provide the standard ASN.1 syntax rules and the CASN1 syntax rules in BNF. Appendix C, Appendix D and Appendix E demonstrate some examples of using CASN1 and the ED library to encode/decode data-values of ASN.1 types.
Chapter 2

The OSI ASN.1 and BER

This chapter briefly introduces the ISO standard ASN.1 and BER. It also points out the differences between CCITT recommendation X.409 and ISO standards ASN.1 and BER. The BNF specification of ASN.1 syntax is included in Appendix A. Interested readers should refer to the ISO documents cited in the bibliography for more complete descriptions of the standards.

2.1 Abstract Syntax Notation One (ASN.1)

Abstract Syntax Notation One (ASN.1) is a language which enables both complex types to be defined and the values of these types to be specified without concern regarding how instances of these types are represented during data transfer. In many aspects, ASN.1 is similar to the data type definitions in conventional programming languages, such as Pascal, C, and Ada. It has a formal syntax for defining and naming types, as well as mechanisms allowing users to build arbitrarily complex constructor types from simpler ones.

As specified in ASN.1, a type consists of a collection of values. One specific instance of a type is called a value of the type. More generally, a value or a type often consists of several simpler values or types, together with the relationships between them. ASN.1 defines seven primitive types: INTEGER, BOOLEAN, REAL, ENUMERATED, BIT STRING, OCTET STRING, NULL
and OBJECT IDENTIFIER. The values of these types are also defined by ASN.1. In ASN.1, there are five constructors for defining more complex types: SEQUENCE, SET, SEQUENCE OF (SET OF), CHOICE, and SUBTYPE. These constructors are applied according to the following rules:

1. Given an ordered list of existing types, a value can be formed as an ordered sequence of values, by choosing one from each type in the list. The collection of all possible values obtained in this way is a new type — SEQUENCE. If the tags of these types and of any immediately following types in the list are distinct, SEQUENCE can be extended to allow omissions of some values from the list.

2. Given a list of distinct existing types, a value can be formed as a (un-ordered) set of values, one from each type in the list. The collection of all possible values obtained in this way is a new type — SET. SET can be extended to allow omissions of some values.

3. Given a single existing type, a value can be formed as a sequence or a set by choosing zero, one or more values of the given type. The collection of all possible values obtained in this way is a new type — SEQUENCE OF or SET OF.

4. Given a list of distinct types, a value can be chosen from any one of them. The set of all possible values obtained in this way is a new type — CHOICE.

5. Given a type, a new type can be formed as a subset of it by using some structure or ordering relationship among the values, resulting in a new type — SUBTYPE.

Any types defined by one of the above five rules is a structured type. Obviously, any value of these structured types are composed of the values of their component types.

To correctly interpret a structured data-value, it is always necessary to know the type of the value. In a high level programming language, the type of a data-value can be found by looking
at the variable declaration. In a BER octet-sequence transmitted on network links, the type of a data-value can be represented by certain octets and those octets are also transmitted as an integral part of the data-value.

In ASN.1, every type is assigned a tag. A value is always associated with the type-tag to indicate the type of the value. The tag of a type can be defined either by ASN.1 or by users. Four tag classes are specified in ASN.1 and each of them represent certain kind of types:

1. The **universal** class defines tags that are only used as specified in ASN.1. Each universal class tag is assigned either to a single primitive type or to a single constructor type.

2. The **application** class defines tags that are assigned to types defined by other standards or recommendations. Within a particular standard, an application class tag is assigned to only one type.

3. The **private** class defines tags that are never assigned by ISO standards or CCITT recommendations.

4. The **context-specific** class defines tags that are freely assigned within any use of ASN.1, and are interpreted according to the context in which they are used.

A tag consists of **class**, **form** and **identifier code** components. A tag's **class** component is one of the above four defined classes. Within the same class, tags are distinguished by the **identifier code** of the tag which may be any positive integer. A tag's **form** component indicates how the data is represented during transferring, either in primitive form for most primitive types or in constructor form for structured types and some primitive types. Tags are mainly intended for machine use rather than the user. It is common to assign a tag to multiple different types. Two
types can be distinguished by their content when their tags are the same. A user of ASN.1 may choose to assign distinct tags to two occurrences of the same type, resulting in two distinct types. This can be useful, for example, to distinguish which choice has been made in the case of the choice type.

ASN.1 defines a type by one of the seven primitive types or one of the five constructors and assigns a tag to each of the defined types. The types differ by their structures or by their tags. By assigning different tags to OCTET STRING, ASN.1 defines eight character string types: NUMERICSTRING, PRINTABLESTRING, TELETEXSTRING (T61STRING), VIDEOTEXSTRING, VISIBLESTRING (ISO646STRING), IA5STRING, GRAPHICSTRING and GENERALSTRING. By applying different structure compositions, ASN.1 defines four useful types: GENERALIZED TIME, UNIVERSAL TIME, EXTERNAL and OBJECT DESCRIPTOR. From all these existing types, more complex types may be defined.

ASN.1 can be applied whenever it is necessary to define the abstract-syntax of information. It is particularly applicable to specifying application protocol-standards. ASN.1 is used as a semi-formal tool for defining protocols. The use of the notation does not necessarily preclude ambiguous specifications. The user is responsible for ensuring that the specification is not ambiguous.

ASN.1 is supported by a set of encoding rules. One of them is the Basic Encoding Rules (BER). The application of the encoding rules results in a complete specification of the data-value as well as its type during data transfer; i.e., BER transfer-syntax. Figure 2.1 shows an example of encoding an ASN.1 INTEGER with the value of 8824 using BER.
CHAPTER 2. THE OSI ASN.1 AND BER

2.2 Basic Encoding Rules (BER)

The Basic Encoding Rules (BER) are a set of encoding rules that may be applied to values of ASN.1 types to produce a transfer-syntax for those values. BER are used at the time of data transformation by the presentation-service provider. It is implicit in the specification that the encoding rules are also used for decoding the transfer-syntax to identify the data-values being transferred.

According to BER, the complete octet-sequence used to represent the data-value consists of four components which appear in the following order:

1. identifier octets,
2. length octets,
3. contents octets, and
4. end-of-contents octet (eoc).

The identifier indicates the type of the data-value. It encodes the ASN.1 tag of the data-value. It may contain more than one octet if the tag identifier code is greater than 32. The
length determines the end of a data-value transfer-syntax representation. Two forms of length encoding are provided: definite and indefinite. In the definite form, the length octets consist of one or more octets to represent the number of octets in the contents octets. In the indefinite form, the length octets consist of a single octet with a special value indicating that the contents octets are terminated by an end-of-content octet. The contents consist of zero, one or more octets to represent a particular data value that distinguish the data-value from other values of the same type. The data-value in contents is encoded as specified in BER. The end-of-contents is used to determine the end of a data-value transfer-syntax representation. It is optional depending on the length octets. It will be presented only if the value of the length indicates indefinite form. Figure 2.2 shows two structures of a data-value encoded according to BER\textsuperscript{1}.

![Diagram](image)

<table>
<thead>
<tr>
<th>IDENTIFIER-OCTETS</th>
<th>LENGTH-OCTETS</th>
<th>CONTENTS-OCTETS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>&quot;Represents number of octets in contents octets&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IDENTIFIER-OCTETS</th>
<th>LENGTH-OCTETS</th>
<th>CONTENTS-OCTETS</th>
<th>END-OF-CONTENTS OCTETS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>&quot;Indicates that the contents octets are terminated by end-of-contents octets&quot;</td>
<td>&quot;Indicates that there are no further encodings in the contents octets&quot;</td>
</tr>
</tbody>
</table>

Figure 2.2: Structure of BER Transfer Syntax

\textsuperscript{1}This figure is from [9].
2.3 Differences between X.409 and ISO ASN.1/BER

ASN.1 defined in ISO 8824 and BER defined in ISO 8825 are technically aligned with the relevant parts of CCITT X.409 which predates ASN.1 and BER. CCITT X.409 defines an abstract-syntex notation and encoding/decoding rules in a single document. As listed below, ASN.1 has several extensions to the abstract-syntex notation which are not included in X.409:

- **IMPORT** and **EXPORT** types among modules.
- **TAG DEFAULTS** are added to define the default tag of type definitions in a module.
- **SUBTYPE** notation is added to the type-definitions.
- **ANY DEFINED BY** is added to the **ANY** type-definition.
- Five new types are introduced: **OBJECT IDENTIFIER**, **REAL**, **ENUMERATED**, **OBJECT DESCRIPTOR** and **EXTERNAL**.
- Three more character string types are defined: **VISIBLESTRING** **GRAPHICSTRING** and **GENERALSTRING**.

Along with these extensions defined by ASN.1, BER extends X.409 to include encoding rules for the newly defined types.

The CCITT Recommendation X.208 and X.209 are the CCITT standards corresponding to the ISO 8824 (ASN.1) and 8825 (BER) respectively. CCITT X.208 extends ASN.1 to include the **ENCRYPTED** type. Along with the X.208 extension to ASN.1, X.209 extends BER to include the encoding rules for the **ENCRYPTED** type.
Chapter 3

Data Type Translation between ASN.1 and C

Because the C programming language has been widely used in protocol implementations, it has been chosen as the application programming language in our implementation. For most data-types, translation between ASN.1 and C can be achieved simply by direct mappings. However, there are some ASN.1 types which require the design of special C structures for the translations. Table 3.1 summarizes the basic translation between ASN.1 types and C types. They are further explained in the following sections.

3.1 Primitive Types

The ASN.1 types INTEGER, REAL, ENUMERATED, NULL, SEQUENCE { ... } and SET { ... } are directly mapped to the corresponding C types int, float, enum, int and struct { ... }. To reduce memory consumption, the ASN.1 type BOOLEAN is translated into a C type bool which is defined as an unsigned char.

1In this thesis, the terminologies C type, C-definition and C-structure are used interchangeably to denote the C-translation of ASN.1 type definitions. Also, a word in small caps font, such as INTEGER, indicates that it is a token in ASN.1 specifications and a word in type-writer font, such as int, indicates that it is a token in C-programs.
<table>
<thead>
<tr>
<th>ASN.1 Type</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>bool</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>ENUMERATED</td>
<td>enum</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>BIT STRING</td>
<td>BITS</td>
</tr>
<tr>
<td>OCT STRING</td>
<td>OCTS</td>
</tr>
<tr>
<td>NULL</td>
<td>int</td>
</tr>
<tr>
<td>OBJECT IDENTIFIER</td>
<td>OID</td>
</tr>
<tr>
<td>SEQUENCE { ... }</td>
<td>struct { ...}</td>
</tr>
<tr>
<td>SET { ... }</td>
<td>struct { ...}</td>
</tr>
<tr>
<td>SEQUENCE OF</td>
<td>LIST*</td>
</tr>
<tr>
<td>SET OF</td>
<td>LIST*</td>
</tr>
<tr>
<td>CHOICE { ... }</td>
<td>struct {</td>
</tr>
<tr>
<td></td>
<td>int choice;</td>
</tr>
<tr>
<td></td>
<td>union { ...} data</td>
</tr>
<tr>
<td>CHARACTER STRING TYPE</td>
<td>OCTS</td>
</tr>
<tr>
<td>UNIVERSAL TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>GENERALIZED TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>EXT</td>
</tr>
<tr>
<td>ENCRYPTED TYPE</td>
<td>BITS</td>
</tr>
</tbody>
</table>

Table 3.1: Data-Type Translation between ASN.1 and C
CHAPTER 3. DATA TYPE TRANSLATION BETWEEN ASN.1 AND C

typedef unsigned long UNIV;

typedef struct BITS {
    struct BITS *next; /* pointer to next BITS node */
    long len; /* number of bits in the bit string */
    UNIV data; /* pointer or value of the bit string */
} BITS;

typedef struct OCTS {
    struct OCTS *next; /* pointer to next OCTS node */
    long len; /* number of octets in the octet string */
    UNIV data; /* pointer to the octet string */
} OCTS;

Figure 3.1: C Structure BITS and OCTS Definition

3.2 Bit String and Octet String Types

According to the ASN.1 definition, the ASN.1 types BIT STRING and OCTET STRING may be in CONSTRUCTOR form. Therefore, they are translated into C types BITS and OCTS respectively which are defined in Figure 3.1.

The next field in BITS is a pointer to a BITS structure where another bit string may be stored. The len field records the length of the bit string (in number of bits). If the length is no longer than 32 bits, the data field contains the value of the bit string starting from the left-most bit. Otherwise, the data field stores a pointer to an octet-string where the bit string value is stored starting from the left-most bit of the first octet. The three fields in OCTS function the same as those in BITS. Notice that the len field contains the number of octets in the octet string and that the data field always keeps an octet-string pointer.
typedef unsigned long UNIV;

typedef struct OID {
    long len; /* number of identifier components < 20*/
    int oid[20]; /* object identifier components array */
} OID;

Figure 3.2: C Structure OID Definition

3.3 Object Identifier Type

The ASN.1 type OBJECT IDENTIFIER is translated into a C type OID as defined in Figure 3.2. The two fields oid and len record the object identifier components and its length. In order to statically initialize the oid field, it is defined as an integer array with 20 components. This is based on the assumption that there are no more than 20 components in any object identifier.

3.4 Any Type

Since an ASN.1 type ANY may represent data of any ASN.1 type, it is simply translated into a C type UNIV which is defined as unsigned long. UNIV can be considered a pointer to any C type.

3.5 Sequence and Set Types

As explained in Chapter 2, the ASN.1 type SEQUENCE/SET contains an ordered/un-ordered list of distinct components ASN.1 type. Both SEQUENCE { ... } and SET { ... } are translated into the C type struct { ... } where the "..." are the C definitions of the components ASN.1 type in the given SEQUENCE or SET type. The field names within the C-structure are the corresponding names of the components of the ASN.1 type. If a component type is OPTIONAL
or is recursively defined, its corresponding field in the C-structure will be a pointer to the C-definition of that component ASN.1 type. If a component is COMPONENTS OF, its C translation will contain more than one field, one for each of the component types’ C-translations.

In order to simplify the data encode and decode procedures, a multi-level structured type is serialized before being translated. Each of the structured component types is defined as a new type. The corresponding field of the structured type will be re-defined by the new type name. Appendix C contains an example of serializing a multi-level structured ASN.1 type.

### 3.6 Sequence of and Set of Types

The ASN.1 types SEQUENCE OF and SET OF are translated into a C type LIST which is defined in Figure 3.3.

```c
#define unsigned long unit_32;
#define unsigned long UNIV;

typedef struct LIST_ITEM {
    UNIV *item;  /* pointer to the list item */
    struct LIST_ITEM *next;  /* pointer to next list item */
} LIST_ITEM;

typedef struct LLIST {
    uint_32 count;  /* length of the list */
    LIST_ITEM *top;  /* pointer to the first list item */
    LIST_ITEM *next;  /* pointer to current list item */
} LLIST, *LIST;
```

Figure 3.3: C Structure LIST Definition

Each list item contains the data-value of one component. The item field in the LIST_ITEM is a pointer to the item data value and the next field is a pointer to another list item. In the
LIST structure, the count field records the number of items in the list and the other two fields are pointers to the first and the current item. LIST has been defined as an abstract data type. A number of operations on it have been implemented. For example, ListCreate() creates a new list and ListAppend() appends a new item to a list.

3.7 Choice Type

The ASN.1 type CHOICE is also translated into a C type struct. Since it can be only one of the choices at any instant, the field data is defined as a union to store the value of the current choice. The field choice indicates the type of choice by containing the type-tag. In order to simplify the structure of a CHOICE type encoder/decoder, sub-field choice is defined as a C type int. It represents an ASN.1 type-tag as a 32 bit integer. Three tag components, class, form and identifier-code, are integrated into the 32 bits. The tag-class is represented by the left-most two significant bits. The tag-form is represented by the left third bit. This is similar to the tag encoding in BER. However, the remaining 29 bits, starting at the fourth bit, represent the tag-identifier-code as a 29 bit integer. This is based on the assumption that a tag identifier-code is no less than 0 and no greater than $2^{29} - 1$. The ED library utility routine PackTag() converts the three components of an ASN.1-tag into such a “tag-integer” for the choice field. Another ED library utility routine GetTag() extracts the ASN.1 tag from the BER octet sequence and converts it into a “tag-integer”. Appendix D shows an example of the translation from an ASN.1 type CHOICE into a C-structure.
typedef struct TIME {
    int year; /* year : 0 .. "(19** or **) */
    int month; /* month : 1 .. 12 */
    int day; /* day : 1 .. 31 */
    int hour; /* hour : 0 .. 23 */
    int minute; /* minute : 0 .. 59 */
    float second; /* second : 0 .. 59 */
    float diff; /* diff between local and standard time */
    int zone; /* time flag: (0,1,2,3,4) */
} TIME;

#define UTC_Z_TIME 0 /* UTC time with Z */
#define UTC_D_TIME 1 /* UTC time with differential */
#define GNL_Z_TIME 2 /* Generalize time with Z */
#define GNL_D_TIME 3 /* Generalize time with differential */
#define GNL_L_TIME 4 /* Generalize local time */

Figure 3.4: C Structure TIME Definition

3.8 UTC Time and Generalized Time Types

The ASN.1 types UNIVERSAL TIME and GENERALIZED TIME are both translated into a C type TIME which is defined in Figure 3.4. The second field in TIME is a float type which permits the TIME structure to represent time to the precision of one hundredth of a second. The diff field contains the difference between the local time the standard time (Greenwich time). It is also a float type, with the integer part containing the differential in minutes and the fractional part containing the differential in seconds. The zone field indicates the type of the time value as denoted by the five constants defined in Figure 3.4.

3.9 External Type

The ASN.1 type EXTERNAL is translated into a C type EXT which is defined in Figure 3.5. Field dref is equal to NULL if there is no direct reference. Field iref may also be NULL if the
typedef struct OID {
    int len;  /* number of oid components < 20 */
    int oid[20];  /* object identifier components */
} OID;

typedef struct CODING {
    int choice;
    union {
        #define single_asnl_$tag 0x80000000
        OCTS single;  /* single ASN.1 type */
        #define octetAligned_$tag 0x80000001
        OCTS octet;  /* octet aligned */
        #define arbitrary_$tag 0x80000002
        BITS arbitrary;  /* arbitrary */
    } data
} CODING;

typedef struct EXT {
    OID *dref;  /* direct reference (optional) */
    int *iref;  /* indirect reference (optional) */
    OCTS *value;  /* data value descriptor (optional) */
    CODING ed;  /* Encode policy choice */
} EXT;

Figure 3.5: C Structure EXT Definition

indirect reference is not available. Field value contains the data-value descriptor if there is any. Field ed is of C type CODING which contains the chosen encoding policy.

3.10 Defined Tagged Types

Since in ASN.1, all character string types (NumericString, PrintableString, TeletexString, VideotextString, VisibleString, IA5String, GraphicString and GeneralString) are defined as tagged OCTET STRING types, they are all translated into the C type OCTS. For the same reason, the ASN.1 type OBJECT DESCRIPTOR is translated into the C type
OID and the ASN.1 type ENCRYPTED is translated into the C type BITS.
Chapter 4

The ED Library Design and Implementation

The ED library is an implementation of the Basic Encoding Rules (BER) in C on UNIX 4.2 BSD. It is based on the ISO 8825, ISO 8825 and their addendums. The CCITT Recommendation X.208, X.209 are also referenced.

4.1 The ED Library Design

The ED library includes 38 routines for encoding and decoding ASN.1-types. The encoding routines translate data-values from C into BER octet-sequences. The decoding routines translate data-values from the BER octet-sequences back to C. Figure 4.1 shows the role of the encoding and decoding routines.

The input to the ED library encoding routines and the output of the ED library decoding routines are data of the C-types listed in Table 3.1. The input to the ED library decoding routines and the output of the ED library encoding routines are octet-sequences in BER transfer-syntax. The input octet-sequences to the decoding routines may be stored either in main memory (as a contiguous octet-string) or on disk (as a file). A user must indicate this when the
octet-sequence is passed to a decoder as a parameter of type IOP. The output octet-sequence of an encoding routine is stored using an IDX structure which can be serialized into a contiguous octet-string by applying the ED library utility routine serIDX(). The definitions of the C structures IDX and IOP are in Section 4.2.1.

The ED library routines are classified into four classes: primitive, constructor, utility and memory. The primitive class includes 18 routines which directly encode and decode data-values of primitive ASN.1-types. The constructor class includes 4 routines for constructing the encoders and decoders of structured ASN.1-types. The utility class includes 9 routines which are used to support the structured type encoders and decoders. The memory class includes 11 routines for the ED library sub-memory system operation.

For any data-type defined by ASN.1, one can construct an encoder and a decoder using the ED library routines to translate its data-value between C and the BER transfer-syntax. The user manuals for the ED library routines is in [20].

4.1.1 The ED Library Utility Routines

There are 10 utility routines in the ED library: Encode_tag(), Decode_tag(), PackTag(), GetTag(), TestTag(), Encode_eoc(), Decode_eoc(), serIDX(), Loadbits() and Loadocts().

Routine Encode_tag() encodes an ASN.1 tag from its three components into the BER
octet sequence. Decode_tag() decodes an ASN.1-type tag from the BER transfer-syntax and compares it with a given tag. Routine PackTag() integrates the three components of an ASN.1 tag into a "tag-integer". Routine GetTag() extracts the tag from a BER octet-sequence and converts it into a "tag-integer". TestTag() compares a tag in a BER octet-sequence tag with a given tag and reports whether they match. serIDX() takes an IDX structure as input and serializes its buf fields into a contiguous octet-string. Routine Encode_eoc() creates an IDX structure and stores the end-of-content octets in it. Routine Decode_eoc() decodes the end-of-content octets from a BER octet-sequence. Loadbits() takes a BITS index structure as input and loads the data-value from disk into main memory. Loadocts() takes an OCTS index structure as input and loads the data-value from disk into main memory. The user manuals for these routines are in [20].

4.1.2 Primitive Type Encoding and Decoding

As shown in Table 4.1.2, an encoding routine and a decoding routine exist in the ED library for each of the listed ASN.1-types. These routines are used directly to convert data-values between C and BER transfer syntax. Detailed information about these routines may be found in their user manuals [20].

4.1.3 Structured Type Encoding and Decoding

To encode or decode the data-value of a structured ASN.1-type, such as SEQUENCE { ... }, SET { ... }, SEQUENCE OF or SET OF, one has to construct the encode and decode routines according to the data structures of the given data-value. The ED library constructor class routines Encode_struct_beg(), Encode_struct_end(), Decode_struct_beg() and Decode_struct_end() are used in constructing these routines.
CHAPTER 4. THE ED LIBRARY DESIGN AND IMPLEMENTATION

<table>
<thead>
<tr>
<th>ASN.1 Type</th>
<th>Encode Routine Name</th>
<th>Decode Routine Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>Encode_bool()</td>
<td>Decode_bool()</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Encode_int()</td>
<td>Decode_int()</td>
</tr>
<tr>
<td>BIT STRING</td>
<td>Encode_bits()</td>
<td>Decode_bits()</td>
</tr>
<tr>
<td>OCTET STRING</td>
<td>Encode_octs()</td>
<td>Decode_octs()</td>
</tr>
<tr>
<td>NULL</td>
<td>Encode_null()</td>
<td>Decode_null()</td>
</tr>
<tr>
<td>OBJECT IDENTIFIER</td>
<td>Encode_oid()</td>
<td>Decode_oid()</td>
</tr>
<tr>
<td>UNIVERSAL TIME</td>
<td>Encode_UTCTime()</td>
<td>Decode_UTCTime()</td>
</tr>
<tr>
<td>GENERALIZED TIME</td>
<td>Encode_GNLTime()</td>
<td>Decode_GNLTime()</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>Encode_ext()</td>
<td>Decode_ext()</td>
</tr>
</tbody>
</table>

Table 4.1: The ED Library Primitive Class Routines

When encoding a structured ASN.1-type, `Encode_struct_beg()` is called first to encode the tag and to set up the structured type encoding environment. Then, the encoder for each component type is called one after another to encode the data-values of the corresponding component types. If a component is a primitive type, its encoder is one of the ED library primitive class routines. If a component type is a constructor type, its encoder will be constructed in the same way as constructing the encoder for a structured type. If a component type is `OPTIONAL`, its encoder is called only when the input data contains a value. Each BER octet-sequence of a component type is stored in one IDX structure. A stack `STK` records the IDX structure pointers of all component types. After the entire structured type is encoded, routine `Encode_struct_end()` is called to link all the component type IDX structures one after another as they were defined. `Encode_struct_end()` calculates the structured type content length and encodes the content length octets according to the content length form indicated by a parameter. For the definite form, an IDX structure is created to store the encoded content length octets and the IDX structure is inserted right after the IDX structure containing the type tag. If the length form is indefinite, both the content length octets and the end-of-content octets are
stored in IDX structures and linked into the output IDX structures. Constructing the decoder for a structured type is similar to construct its encoder. Routine Decode_struct_beg() is called first. It validates the tag and sets up the decoding environment. Then, the decoders of the component types are called one after another to decode the data values of the corresponding element types. The primitive component types are decoded by the ED library primitive class routines. The structured component types are decoded by the corresponding decoders which are constructed in the same way as constructing the decoder for a structured type. If the component type is OPTIONAL, the ED library utility routine TestTag() is called to test the tag and determine whether to call its decoder. A stack called LENS records the start-address and the content length of the input BER octet-sequence. After the entire data-value of a structured type is decoded, Decode_struct_end() is called to pop LENS and verify the end-address of the BER octet-sequence. The decoded data-value is stored in the corresponding C structure whose memory space is allocated by the decoder. Users are responsible for releasing the memory after using it. An example of encoding and decoding a structured type is included in Appendix C.

For a CHOICE type, the encoder and decoder is constructed of the encoders and decoders of its choices in a switch statement. Since each of the choices has a distinctive tag, the tag of the input data-value can be used to determine the choice and select the corresponding encoder to encode the data-value. Appendix D shows an example of encoding and decoding a CHOICE type. The “tag-integer” of each choice is defined as a constant with the name of type-name followed by the choice-name followed by a “_tag”.

4.1.4 Tagged Type Encoding and Decoding

The ASN.1 Tagged type is defined as a type with an assigned tag. The manuals in [20] showed that the encoding and decoding routines take the tag as the input parameters. There-
fore, as long as the assigned tag is used for the tag parameters, a tagged type can be encoded by the same encoder of its original type. For the same reason, a tagged type decoder is the same as its original type decoder.

The ASN.1 character string types are defined as tagged OCTET STRING type. Therefore, they all use the routines Encode_octs() and Decode_octs() to encode and decode their data-values with the specific tags as parameters. For the same reason, the ASN.1 type ENUMERATED uses Encode_int() and Decode_int() and OBJECT DESCRIPTOR uses Encode_octs() and Decode_octs() as their encoders and decoders respectively. The ENCRYPTED type, defined in the CCITT Recommendation only, is a tagged BIT STRING type. Its data-value can be encoded/decoded by the ED library routine Encode_bits() and Decode_bits().

4.1.5 Special Feature of Decoding

Usually the ED library decoding routines decode the entire data-value from the input octet-sequence into the corresponding C structure in main memory. However, there is an alternative for decoding data of BIT STRING or OCTET STRING type. When the input transfer-syntax is in CONSTRUCTOR form of BIT STRING or OCTET STRING type, it usually contains a large data-value. It may not be necessary to decode the entire constructor into main memory. Therefore, when the input data is from a file and is in CONSTRUCTOR form of BIT STRING or OCTET STRING type, Decode_bits() and Decode_octs() only decode the structure of the input data-value and store indices in the C structures BITS or OCTS. The indices are based on the input file. The data field in C structure BITS or OCTS records the input file name and the len field records the data offset in the file. These BITS or OCTS indices can be converted into real data-values by applying the ED library utility routine Loadbits() or Loadocts() to load the whole data-value from the file into C structure BITS or OCTS in main memory. The implementation details of
BIT STRING and OCTET STRING decoding and loading may be found in Section 4.2.2.

The advantage of this special feature is to provide users with a means of decoding data in an efficient way. Since decoding a large data-values takes a long time and consumes a large amount of memory in copying the data-values, it is much more efficient to leave the data-value in the file and only extract its data-structure as needed.

4.1.6 The ANY Type Encoding and Decoding

The ASN.1 ANY type is defined as a representation of any types defined by ASN.1. Both its data-type and its data-value are indefinite and non-determined. Therefore, there is no particular encoder or decoder for ASN.1 ANY type. However, since any instance of ANY type is an ASN.1-type, one can always use the ED library routines to construct the encoder and decoder for a particular instance of ANY type.

4.1.7 The REAL Type Encoding and Decoding

The ED library should have routines Encode_real() and Decode_real() to encode and decode data-values of ASN.1 REAL type. However, in the current implementation it is not supported.

4.2 The ED Library Implementation

The ED library has been implemented in the C on UNIX 4.2 BSD. The source consists of about 3,400 lines of C code. The binary is about 10K bytes.
CHAPTER 4. THE ED LIBRARY DESIGN AND IMPLEMENTATION

4.2.1 The ED Library Data Structures

Data Structure IOP

The input BER octet-sequence to a decoding routine may be stored either in main memory or on disk. Users must indicate this in the parameter of type IOP when passing a BER octet-sequence as input into a decoding routine. The structure IOP is defined in Figure 4.2. The b field stores the pointer to the input octet-sequence if the octet-sequence is from main memory, otherwise it is NULL. If the input octet-sequence is from a file, the f field stores the file pointer and the fname field stores the file name. Otherwise both f and fname are NULL. All decoding routines test the f field first. If it is NULL, the input octet-sequence will be considered as a contiguous octet-string from main memory.

```c
typedef unsigned char byte;

typedef struct IOP
{
  byte  *b;           /* pointer to an octet string */
  FILE  *f;           /* pointer to a file */
  byte  fname[30];   /* file name with max len=30 */
} IOP;
```

Figure 4.2: C Structure IOP Definition

Data Structure IDX

The data structure IDX is defined in Figure 4.3. It is designed to store the output octet-sequence of the encoding routines. It is also used in the ED library sub-memory system. The IDX buf field stores a pointer to an octet-string. The len field stores the length of the octet-

typedef unsigned char byte;

typedef struct IDX {
    byte* buf; /* pointer to an octet string */
    long len; /* length of the octet string buf */
    struct IDX *next; /* pointer to next IDX node */
} IDX, *ptrIDX;

Figure 4.3: C Structure IDX Definition

string in the buf field. The next field points to another IDX structure.

When encoding a structured ASN.1-type, one does not know its content length until the entire structure is encoded. According to BER, the content length octets should be in front of the content octets in the encoded BER octet-sequence. Because it is possible that the content length octets consist of more than one octet, the encoded octet-sequence should be stored in a data structure whose space can be dynamically extended. The IDX structure, as a linked list, can dynamically extend its space using the next field.

For each structured ASN.1-type, its encoder calls routine Encode_tag() to encode the tag and store the encoded octets in one IDX structure. All encoded component octets are stored in the IDX structures and these IDX structures are linked together as they are defined. After the entire data structure is encoded, an IDX structure is created to store the content length and inserted into the encoded octet-sequence right after the IDX structure containing the tag.

The octet-sequence stored in the buf fields of an IDX structure link can be serialized into a contiguous octet-string by applying the ED library utility routine serIDX().


```
IDXptr S[STACK_SIZE];
IDX  **STK = (IDX *) S;  /* stack for IDX pointers */

long  S_LEN[STACK_SIZE];  /* stack for octet sequence */
long  *LENS = (long *) S_LEN;  /* address and length */
```

Figure 4.4: The ED Library Stack Definition

The ED Library Internal Stacks

There are two internal stacks in the ED library, STK and LENS. Their definitions are in Figure 4.4. Stack STK records the IDX structure pointers during a structured data encoding. Stack LENS records the input octet-sequence address and content length during a data decoding. Both STK and LENS are implemented as abstract data types. A number of operations on the two stacks have been implemented. XINIT() initiates stack STK. XPUSH() and XPOP() manipulate the STK pointer and data-values. XTOP() or XBOTTOM() reads the data-value from STK top or bottom respectively. Operation LINIT(), LPUSH(), LPOP(), LTOP() and LBOTTOM() are the corresponding functions for LENS. LPLUS() increments the LENS stack pointer by the value of a given parameter.

4.2.2 The ED Library Routines

The implementation of most ED library routines is straightforward. The primitive routines simply translate data of certain ASN.1-types between the C representations and the BER octet-sequences. The constructor routines are supported by some internal data structures.
Encode_struct_beg() and Encode_struct_end()

Encode_struct_beg() sets up a structured type encoding environment. It turns on the structured type flag STRUCT, which is a global variable, to indicate that the following procedures are encoding the structured type components. It creates an IDX structure to store the tag and pushes the pointer to the structured type onto stack STK. The encoded component type octet-sequences are also stored in IDX structures and their pointers are also pushed onto stack STK. When the entire data-value of a structured type has been encoded, Encode_struct_end() is called to complete the structured type decoding. Encode_struct_end() pops stack STK and links all the element-type IDX structures together as they are defined. It also calculates the structured type content-length by summing the octet-sequence length of all element-types. Depending on the required length form which is indicated by one of its parameters, Encode_struct_end() creates new IDX structures for storing the content length octets and inserts these IDX structures into the encoded octet-sequence. The structured type flag STRUCT is turned off when Encode_struct_end() returns.

Decode_struct_beg() and Decode_struct_end()

Decode_struct_beg() sets up a structured type decoding environment. It checks the tag of the input BER octet-sequence, extracts its content length and pushes the input octet-sequence address and the content length onto stack LENS. After the entire data-value of a structured type is decoded, the routine Decode_struct_end() is called to complete the decoding. Decode_struct_end() pops the stack LENS and validates the octet-sequences end-address by comparing it with the sum of the corresponding starting address and content length. Error messages will be printed if they do not match.
Loadocts() and Loadbits()

As mentioned in Section 4.1.5, routines Decode.octs() and Decode.bits() use a special way of decoding ASN.1 BIT STRING or OCT STRING data-values when the input BER octet-sequence is in CONSTRUCTOR form and is from a file. They only decode the data structure and save indices in the C-type OCTS or BITS. When OCTS or BITS stores the data structure indices, each node stores the index of one piece of the string. Its data field stores the input file name and its len field stores the offset of the string value in the input file. Later, a string data-value can be load from the input file into a C-type BITS or OCTS respectively by applying the ED library utility routines Loadocts() or Loadbits().

Routine Loadbits() and Loadocts() take a BITS or OCTS index structure as a parameter, respectively. They open the data file whose name is stored in the BITS or OCTS data field, move the file pointer to the offset indicated by the len field, and load the octet-sequence from the file into main memory. The decoded string value is stored in the data field, replacing the file name, and the string length is stored in the len field replacing the string offset.

Before loading a data-value from a file into main memory, Loadbits() and Loadocts() verify the input BITS and OCTS respectively. If the file specified by the fname field does exist, it is considered a correct index structure. Otherwise, an error message will be printed. Detailed information can be found in the ED library user manuals in [20].

4.2.3 The ED Library Sub-memory System

Since the data encoding and decoding are integral parts of every data transfer, their efficiency affect the overall system performance. Therefore, a great amount of effort has been made to make the ED library implementation efficient. The ED library sub-memory system is an ex-
ample. In the C language, the system calls for memory allocation are very time-consuming. To speed up the ED library, a sub-memory system was built to support memory operations in a more efficient manner.

The sub-memory system is composed of two memory pools, a free IDX pool and a free octet pool. The free IDX pool is a linked list of unused IDX nodes. Their buf fields are NULL. The free octet pool is also a linked list of IDX structure. The buf fields in these IDX nodes point to unused buffer blocks. Routine INITMEM() sets up the sub-memory system and initializes the two pools. The initial memory space of the two pools, 100 IDX nodes for the free IDX pool and 4K octets for the free octet pool, are allocated at compile time. The space in the two memory pools is dynamically extended when necessary.

Routines INITIDX(), NEWIDX(), GETIDX() and FREEIDX() are the sub-memory system operations for IDX nodes. INITIDX() initializes the free IDX pool by resetting the memory space of the IDX linked list. GETIDX() allocates an IDX node to the user. It removes the first node in the free IDX pool and returns its pointer. If the free IDX pool is used up, it calls NEWIDX() to obtain one more block of memory space from system for the free IDX pool. FREEIDX() release IDX nodes by returning them to the free IDX pool.

Routines INITBUF(), NEWBUF(), GETBUF(), FREEBUF() and SQZBUF() are the sub-memory system operations for octets. INITBUF() creates the free octet pool head node and allocates a block of octets to it. GETBUF() allocates memory space to the user. It goes through the free octet pool to get the first-fit memory segment and returns its pointer. If there is no memory segment in the free octet pool large enough, routine SQZBUF() is called to compress them and try again. If there is still no room, the routine NEWBUF() is called to extend the free octet pool by obtaining a new block of octets from the system. FREEBUF() releases memory space by
returning space to the free octet pool.

To reduce the work of system garbage collection, routine \texttt{RESETMEM()} has been implemented to release all memory space allocated by routine \texttt{GETIDX()} and \texttt{GETBUF()} and to reset the sub-memory system. These memory operations are very cheap. Most operations simply modify the pointers in the corresponding memory pools.

Detailed information is in the user manual of the ED library routines [21].

4.3 Using the ED Library Routines

All the ED library routines are C procedures. They depend on two include files, \texttt{defs.h} and \texttt{element.h}, which define the ED library internal data structures and constants.

A user program should include the two header files when using any of the ED library routines. The ED library should be included when loading the user program.
Chapter 5

The CASN1 Design and Implementation

Currently, many protocol standards are specified in a certain protocol specification language, such as ASN.1, Estelle, etc. To implement a protocol, a programmer has to spend a significant amount of effort in translating the protocol definition from the protocol specification language into a protocol implementation language such as C, PASCAL, etc. To achieve the goal of exchanging data in a heterogeneous computing environment, the encoders/decoders for the protocol defined data-types must be constructed as well. However, this type of translation and implementation are usually tedious, time consuming, and error prone. It would be very convenient if a software package could automatically fulfill these arduous work. Motivated by this idea, an ASN.1-C compiler, named CASN1, has been designed. CASN1 compiles a protocol ASN.1-specification into the corresponding C-definitions, and automatically generates the BER encoders/decoders for all compiled data-types. The encoders/decoders generated by CASN1 use the encoding and decoding routines of the ED library. CASN1 has been implemented in C on UNIX 4.2 BSD using YACC and LEX as tools. It is based on ISO 8824 ASN.1-definition, ISO 8825 BER-definition and their addendums. The CCITT recommendation X.208 and X.209
are also referenced.

5.1 The CASN1 Design

CASN1 has been designed and implemented as a three pass compiler. The interfaces and functions of each pass are presented below.

5.1.1 The Interfaces of CASN1

As shown in Figure 5.1, CASN1 has both internal interfaces and the external user interfaces. Users only need to use the external interfaces.

CASN1 External Interface

In the current version, the input of CASN1 is a file containing a single ASN.1 module. For multi-module ASN.1 specifications, CASN1 requires each module to be stored in one file and to be compiled separately.

CASN1 generates four output files: *.defs.c, *.encode.c, *.decode.c and *.init.c, where the * in these file names denotes the input ASN.1 module name\(^1\). File *.defs.c contains the C translation of the input ASN.1 module and the type-definitions of the encode/decode procedures for every data-type. It will be included by the communication software as a header file. File *.encode.c contains the C-code of data-type encoding routines. File *.decode.c contains the C-code of data-type decoding routines. File *.init.c contains the C-code of an initialization routine which will be invoked by the communication software to initialize the environment for encoding and decoding routines.

---

\(^1\) For example, if the input ASN.1 specification module name is VTP, the output files are named in VTP.defs.c, VTP.encode.c, VTP.decode.c and VTP.init.c. Notice that it is the input ASN.1 specification module name which is used, rather than the name of the input file containing the ASN.1 module definition.
CHAPTER 5. THE CASN1 DESIGN AND IMPLEMENTATION

Figure 5.1: The Interfaces of CASN1
Besides the above four C files, CASN1 generates two other compilation diagnosis files pass1 and pass3. They are the execution scripts of the first and third passes of CASN1, respectively. pass3 also includes some statistic information for the input ASN.1 module, such as the name of the input ASN.1 module, the names of all referenced ASN.1 modules and a type table. The type table contains the names, the tags and the tag integer values of all defined types.

While parsing the input data, if any syntax error is detected, CASN1 raises an exception and stops. The exception message reports the error and prints the error line number.

CASN1 Internal Interfaces

CASN1 translates an ASN.1-specification into C-code by three passes. The output of the first two passes are stored in files out1 and out2 which are used as the input to the next pass respectively.

5.1.2 The Functions of CASN1

The first two passes function as pre-processors to simplify the input ASN.1 specification for the third pass.

CASN1 First pass

The first pass takes an ASN.1 module definition in as input and parses it. If any syntax error is detected, CASN1 stops and reports the error with its line number.

In an ASN.1 structured type definition, each component is defined by a named-type or a sub-structure. It may or may not have a reference name. To simplify the generation of data-type encoders/decoders, the first pass serializes the nested structured type-definitions in the

---

2In this thesis, the types of structured type components are referred as component types, sub-types, child types; the reference names of structured type components are referred as field names.
input ASN.1 specification. It defines each sub-structure as a new type and assigns a name to it. An assigned type name consists of a character “T” followed by at least one digit. Then the structured type is re-defined by using the newly assigned sub-type names in the corresponding component definitions. Appendix C shows an example of serializing a nested structured type.

In a structured type definition, if any component field name absents, the first pass will assign a field name to it. An assigned field name consists of a character “r” followed by at least one digit. The reason of filling in all field names is that they are going to be used as the component variable names when the structured type is translated into C.

In order for CASN1 to generate C-code properly, the first pass moves the ASN.1 key word OPTIONAL to the front of the component-type definition. Special lines are inserted between type-declarations to mark type-definition boundaries.

**CASN1 Second pass**

In C language, type declarations obey the bottom-up order, i.e., a type has to be defined before being referenced. While in ASN.1, the type declaration order is not clearly specified although many standards adopt the common convention of top-down order, i.e., parent types are specified before their component child-types. Because of the above, CASN1 assumes that the input ASN.1 specification obeys the conventional top-down order in type-declarations. To simplify the type-definition translation from ASN.1 into C, the second pass inverses the top-down type-definition order of the ASN.1-specification to the C-style bottom-up order, according to the special lines among type-declarations which are inserted by the first pass. Violations of top-down type-definition order in the input ASN.1 module result in errors being reported in the third passes.
CASN1 Third pass

The third pass does a various of error checking and generates the C code. It takes the second pass output as the input, parses it and checks the following:

1. a type must be defined before it is referenced, e.g., the component types of a structured type must be defined before the structured type definition;

2. all referenced types must be defined in current module or imported from other modules;

3. each choice of a \texttt{CHOICE} type must have a distinctive tag;

4. a \texttt{SELECTION} type must be defined by a \texttt{CHOICE} type and its identifier must be the name of one of the choices.

5. a \texttt{COMPONENTS OF} type must be defined by a structured ASN.1 type, such as \texttt{SEQUENCE} \{ ... \} or \texttt{SET} \{ ... \}.

If any error is detected, the third pass reports the error with its line number in file \texttt{out1} and aborts the compilation. CASN1 does not handle cross-module references. It prints a warning message for every external type reference to let users deal with them.

If the compilation succeeds, the third pass generates the C-code for data-type definitions and their encoders/decoders in four output files \texttt{*.defs.c}, \texttt{*.encode.c}, \texttt{*.decode.c} and \texttt{*.init.c}. All type-definitions and constants are translated into C in file \texttt{*.defs.c}. Type-tags are presented as comments in C type-definitions. Type names and structured type field names remain unchanged. A BER encoder and a BER decoder are generated for each of the compiled types and are stored in files \texttt{*.encode.c} and \texttt{*.decode.c}, respectively. The ASN.1 comments are translated into C comments and are inserted in files \texttt{*.defs.c}, \texttt{*.encode.c} and \texttt{*.decode.c}. 
For every choice type, the third pass defines the tag integer values of all choice tags as constants in file *.defs.c. The name of such a constant is constructed with the type name followed by the choice name and followed by "$tag".

The third pass translates an ASN.1 value-definition into a C variable-declaration in file *.init.c. The value-identifiers are used as variable names. These variables are statically initialized by the defined values. An ASN.1 value-definition is processed only when its C-translation type can be statically initialized.

5.1.3 Modification to ASN.1 Syntax

The ASN.1 macro-definition can dynamically extend ASN.1 syntax rules. There is nothing to prevent such extension from introducing infinite syntax reductions into existing syntax rules. To handle ASN.1 macro-definitions, an ASN.1 compiler must have the ability to dynamically extend its syntax rules while parsing the ASN.1 specification. It is a very complex problem to design a dynamically extendible parser which guarantees that no infinite reduction occurs. As a result, the current version of CASN1 ignores macro-definitions for simplicity.

In many cases, ASN.1 macro-definitions are used to define operations. After removing macro-definitions from ASN.1, we have introduced operation-definition and error-definition into CASN1 syntax for specifying abstract-operations. Figure 5.2 gives the syntax rules of both definitions. It is part of the BNF definition of CASN1 syntax rules. The complete BNF specification of the CASN1 syntax rules is given in Appendix B.

According to Figure 5.2, an operation is identified by an operation name which is defined as an identifier and is assigned a number. Operation arguments, as well as operation results, are defined by ASN.1-types. Errors of each operation are defined by a list of error identifiers. An error-definition is similar to an operation-definition. An error is identified by an error
OperationDefinition ::= OperationName OPERATION OperationTypeNotation "::=" number

OperationName ::= identifier

OperationTypeNotation ::= empty

| "ARGUMENT" NamedType Result Errors

Result ::= empty

| "RESULT" NamedType

Errors ::= empty

| "ERRORS" "{" ErrorNames "}"

ErrorNames ::= empty

| IdentifierList

ErrorDefinition ::= ErrorName "ERROR" ErrorTypeNotation "::=" number

ErrorName ::= identifier

ErrorTypeNotation ::= empty

| "PARAMETER" NamedType

NamedType ::= identifier Type

| Type

IdentifierList ::= identifier

| IdentifierList "," identifier

Figure 5.2: BNF of \texttt{OPERATION} Definition and \texttt{ERROR} Definition
name which is defined as an identifier and is assigned a number. The error parameters are defined by ASN.1-types.

For operation definitions and error-definitions, CASN1 translates operation arguments, results and error parameters into the corresponding C type definitions in file *.defs.c. Their encoders and decoders are also generated in files *.encode.c and *.decode.c. As specified in Figure 5.2, operation names and error names are defined as numbers in file *.defs.c. For each operation, the operation name is used as the index for locating the operation argument encoders/decoders in tables EncodeArgument and DecodeArgument, the operation result encoders/decoders in tables EncodeResult and DecodeResult, and the operation error names in table ErrorIDs. The error names in table ErrorIDs can be used as indices to locate the error parameter encoders/decoders in tables EncodeError and DecodeError.

Besides ignoring macro-definitions and extending operation definition and error definition, some equivalent changes have also been made when using ASN.1 syntax definitions in CASN1 in order to minimize syntax definition conflicts. These may be found by comparing the BNF of ASN.1 syntax definitions and CASN1 syntax definitions included in Appendix B and Appendix A.

5.2 The CASN1 Implementation

The ASN1-C compiler, CASN1, has been implemented in C on UNIX 4.2 BSD with YACC and LEX tools. The source consists of about 5,600 lines of C program and the binary is about 34k bytes.
5.2.1 The First pass

The first pass is a parser for the ASN.1 specification. It pre-processes the input ASN.1 specification and generates the input data for the second pass in the file out1.

The first pass maintains two internal objects: a data buffer srcbuf and a stack stk. All tokens from the lexical analyzer are stored in srcbuf while parsing the input data. Whenever a type-definition has been parsed, either as a top-level type-definition or as a component type definition within a structured type-definition, the data in srcbuf is transferred to stk and srcbuf is re-initialized. A top-level type definition is always pushed onto the bottom level of stk — level 0. A component type-definition is pushed onto upper levels of stk. If a component-type is also a structured type, its children type-definitions are pushed onto the one level upper above its parent type-definition. If a component-type is optional, the key word optional is moved to the front of the component type definition while transferring the data from srcbuf to stk. After a complete type-definition is parsed, the first pass pops off stk to generate the output data in file out1. Each of the structured component types is defined as a new type and is assigned a new type name. The corresponding component type-definitions within the parent type-definitions are replaced by the newly defined component type names.

To fill the absent field names of a structured type and to assign newly defined component-type names, one field of stk has been defined as an integer to count the component-type numbers. A field name is constructed by a character “r” followed by the pointer number of stk and then followed by the component type number. A newly defined component-type name is constructed in the similar manner: a character “T” followed by the stk pointer number and then followed by the component-type number.

The first pass replaces all hyphens (“-”) in the input data by under-lines (“_”) because C
language does not allow hyphens ("-") in type or variable names.

To keep track of the input data, a variable `LineCounter` is defined in the lexical analyzer. Whenever a token of `character return` is analyzed, the `LineCounter` is increased by one. The value of `LineCounter` is used for reporting error positions in exception messages.

The last task of the first pass is to insert a special line,

```
---<<<<<<<<<--
```

between every two type-definitions in the input ASN.1 module. Each pair of special lines marks the boundary of a type-definition in file `out1`.

The lexical analyzer and the parser of the first pass are implemented with the UNIX tools LEX and YACC. The parser has 466 states and consists of 287 grammar rules, with 86 terminals and 140 non-terminals. The source consists of about 2,000 lines of C-code and the binary is about 12K bytes.

5.2.2 The Second pass

The second pass inverses the type definition order of the ASN.1 specification. It is implemented as a simple C program. It takes file `out1` as input and reads the data line by line. The type-definition between each pair of the special lines which were inserted by the first pass is viewed as a single data block. The offsets of these data blocks are recorded in a list. Then the second pass copies these data blocks into file `out2` in the inverse order using the recorded offsets.

The second pass implementation consists of 154 lines C program which is about 3.3K in binary.
5.2.3 The Third pass

The third pass does the "real" translation from the ASN.1 specification into C-definitions and generates the encoders and decoders for all compiled types. It takes the pre-processed ASN.1 specification in file out2 as its input, and generates the C-code in four files: *.defs.c, *.encode.c, *.decode.c and *.init.c.

The CASN1 third pass defines several internal data structures for generating C-definitions and their corresponding encoders and decoders:

State Stack

The state stack stk is defined for recording the information of type-definitions, including type name, tag, structure flag, etc. The stack pointer starts from level 0 — the stack bottom, and is increased whenever a structured type is parsed. The stack is used when generating the C-definitions and encoders/decoders for the compiled data-types.

Type Table

The type table is an array of records for recording all compiled type names, tags, and offsets in the files *.defs.c, *.encode.c and *.decode.c. It is used for detecting redundant type names and for checking type reference relations. The offsets are used for generating the C code for ASN.1 types Selection and Components of.

Reference Type Table

The reference type table is an array for recording referenced type names in the ASN.1 module. It is used for detecting undefined reference types.
Module Table

The module table is an array for recording module names of the input module and all referenced modules.

Operation Table

The operation table is an array of records for recording abstract-operation names, numbers and encoding/decoding routine names for operation arguments and results. It is used for generating the system initialization routine in output file *.init.c.

Error Table

The error table is an array of records for recording error names, numbers, and the error parameter encoding/decoding routine names. It is used for generating the system initialization routine in output file *.init.c.

With these internal data structures, the third pass does the various of error checking and generates C-code for data-type definitions and their encode/decode routines. It parses the input data, stores the input module name and all IMPORTed module names in the module-table. It also stores the IMPORTed type names in the type-table. For a type-definition, the type-tag, form (either EXPLICIT or IMPLICIT) and structure (either primitive or constructor) are recorded in the state stack stk. Similar to the first pass, a top-level type definition is pushed onto the bottom-level of stack stk and its component type definitions are pushed onto the upper levels. Since the first pass already serialized the nested structured type definitions, the level of the state stack stk should never be greater than 3. Whenever a complete type definition has been parsed, the third pass saves the type name, tag and offsets of the C-code in corresponding output files.
in the type-table. Redundant type names are detected by checking through the existing type names in the type-table when adding a new type to it. If a type is defined by a named-type, the named-type will be stored in the reference type-table. If a named-type is an external type\(^3\), the external module name will be stored in the module-table and a warning message will be printed to report the cross-module reference. If the named-type is not an external type and its name cannot be found in the type-table, an error message is printed to report reference error because in C a type must be defined before it is referenced. When parsing a CHOICE type-definition, the third pass checks the type-table to guarantee that all the choices have distinguish tags.

For an ASN.1 SELECTION type-definition, the third pass checks the type-table to guarantee that it is defined by a CHOICE type and the IDENTIFIER is one of the choices. Its C-code are generated by copying the corresponding choice's C-translation from files *.defs.c, *.encode.c and *.decode.c. The file offsets of the choice can be found in the type-table. An ASN.1 COMPONENTS OF type-definition is translated in the similar way. The third pass checks the type-table to guarantee that it is defined by a structured type and then generates its C-code by copying the C-code of all components from files *.defs.c, *.encode.c and *.decode.c. Again, the file offsets of the component-types can be found in the type-table.

A recursively structured type-definition can also be detected by checking through the type-table. A recursive component-type, as well as the OPTIONAL component-type, is translated into a pointer of the corresponding C type.

For an OPERATION-definition, the third pass stores the operation name, number, the encoder and decoder names of the operation arguments and results, and the operation error identifiers in the operation table. It generates C-definitions for operation argument and result types in file

---

\(^3\) *External type* is a type defined in another ASN.1 module.
*defs.c*. It also generates the encoders and decoders for the argument and result types in files
*encoder.c* and *decoder.c*. The third pass does the similar things for an **ERROR** definition.
It stores the error name, number and the names of error parameter encoders/decoders in the
error table and generates C-definitions for error parameter types in file *defs.c*. The encoding
and decoding routines for the error parameters are also generated in files *encoder.c* and
*decoder.c*.

After an complete input ASN.1 module is parsed, the third pass generates a system ini­
tialization routine **INITSYS()** in file *init.c*. It checks the operation table and error ta­
table to construct routine **INITSYS()** for initializing system data structures **EncodeArgument**,
**DecodeArgument**, **EncodeResult**, **DecodeResult**, **EncodeParameter**, **DecodeParameter** and
**ErrorIds**. The third pass prints the type table and the module table in file pass3. It also
checks un-defined reference types by comparing the types in the reference type table and the
type table.

The lexical analyzer and the parser of the third pass are implemented with the UNIX LEX
and YACC tools. The parser has 492 states and consists of 310 grammar rules, with 86 terminals
and 162 non-terminals. The source consists of about 3,600 lines of C-code and the binary is
about 19K bytes.

5.3 Using the CASN1 Output

Since CASN1 syntax is a sub- and super-set of ASN.1 syntax, users may need to modify
the ASN.1-specifications before inputing them to CASN1. All macro-definitions in the ASN.1
specification should be replaced with **OPERATION**-definitions and **ERROR**-definitions. Macro-
definitions may be simply marked out as comments if they can not be properly changed to
CHAPTER 5. THE CASN1 DESIGN AND IMPLEMENTATION

OPERATION and ERROR-definitions.

Command `casn1` at UNIX shell level invokes CASN1. The first argument is used as the input file name. The output files are generated under directory `./out`. After CASN1 compilation succeed, user programs should include the output file `*.defs.c` as a header file and invoke the routine `INITSYS()`, defined in file `*.init.c`, at the beginning to initialize the environment. Data-types defined in the file `*.defs.c` may be used by user programs to declare variables. The encoders and decoders defined in files `*.encode.c` and `*.decode.c` can be invoked to encode and decode the data-value of the corresponding types. User programs should be compiled together with the object-code of files `*.encode.c`, `*.decode.c`, `*.init.c` and the ED library. Figure 5.3 shows how to use the CASN1 output in user programs. Detailed information may be found in [21].
Figure 5.3: Using CASN1 Output
Chapter 6
Performance and Applications

This chapter presents the performance profile of the ED library routines. Because the performance profiling data of related work is not available, the conjecture that the ED library provides better efficiency can only be justified by structural analysis and comparison as discussed in Section 1.3, rather than by comparing performance profiling data.

The results of CASN1 applications are also given in this chapter.

6.1 The ED Library Performance

As mentioned in previous chapters, efficiency has been placed as the top priority in the ED library design and implementation. To profile the performance for the ED library routines, three ASN.1-types are chosen:

1. The OCTET STRING type is a primitive type defined in ASN.1. It is chosen because it is frequently used and has a simple structure.

Performance profiling is a method used to determine the amount of time and storage a program consumed. Interested readers may refer to "Fundamentals of Computer Algorithms", by E. Horowitz and S. Sahni, Computer Science Press, 1978.
2. The **BasicVTItem** type specified in the ISO Virtual Terminal Protocol (VTP) [11] is an ASN.1 **choice** type. It has a very complicated structure. In the VTP implementation [17], the data-structure of the BasicVTItem expands as the input data length increases. This type was chosen because of the special data structure results in special complexity in its data encoding and decoding.

3. The **PersonnelRecord** type is defined as an example in many standards, such as ISO ASN.1, BER and CCITT X.409, X.208 and X209. It is a structured type consisting of three constructors. The definition of PersonnelRecord is included in Appendix E.

The experiments of performance profiling were conducted on a SUN 3/260 under UNIX 4.2 BSD. The tested routines were compiled by the UNIX command *cc -pg*. The pg option automatically generates a file *gmon.out* to record the procedure invocation performance data. A test program was written to call the encoder/decoder 5,000 times for each input. The average number of milliseconds spent per-execution of the encoder (including *SerIDX()* and decoder are shown in the Table 6.1.

In our context, the storage complexity of the encoder/decoder is interpreted as **octet-complexity** which is measured in the number of extra octets used for carrying the type information of the data-value being encoded. The more octets used in carrying the type information for a particular data-type as its value/size grows, the higher the encoder’s octet-complexity is for encoding that data-type, and the less efficient the data-transfer will be when the data-value is transmitted. By varying the input data-value size for each of the above chosen data-types, we have the following observations:

1. The octet-complexity for encoding an **Octet String** is approximately constant as shown in Table 6.2 and Figure 6.1. The stair cases indicate that more octets are needed to
### Input Data type

<table>
<thead>
<tr>
<th>Input Octet</th>
<th>Encode Input Time (msec)</th>
<th>Encode Output Time (msec)</th>
<th>Encoder Time (msec)</th>
<th>SerlDX Time (msec)</th>
<th>Encode Time (msec)</th>
<th>Decode Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.14</td>
<td>0.07</td>
<td>0.21</td>
<td>0.25</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.18</td>
<td>0.11</td>
<td>0.29</td>
<td>0.29</td>
<td>0.26</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0.52</td>
<td>0.34</td>
<td>0.86</td>
<td>0.86</td>
<td>0.64</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>0.94</td>
<td>0.55</td>
<td>1.49</td>
<td>1.49</td>
<td>0.80</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>1.36</td>
<td>0.74</td>
<td>2.10</td>
<td>2.10</td>
<td>1.05</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>2.22</td>
<td>1.20</td>
<td>3.42</td>
<td>3.42</td>
<td>1.21</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>3.74</td>
<td>2.38</td>
<td>6.12</td>
<td>6.12</td>
<td>2.05</td>
<td></td>
</tr>
<tr>
<td>3275</td>
<td>12.74</td>
<td>7.09</td>
<td>19.83</td>
<td>19.83</td>
<td>5.54</td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>19.06</td>
<td>11.27</td>
<td>30.33</td>
<td>30.33</td>
<td>9.59</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>38.30</td>
<td>22.35</td>
<td>60.65</td>
<td>60.65</td>
<td>18.85</td>
<td></td>
</tr>
<tr>
<td>30000</td>
<td>115.78</td>
<td>68.73</td>
<td>184.51</td>
<td>184.51</td>
<td>57.65</td>
<td></td>
</tr>
</tbody>
</table>

### OCTET STRING

<table>
<thead>
<tr>
<th>OCTET STRING</th>
<th>Encode</th>
<th>Encode</th>
<th>Encoder</th>
<th>SerlDX</th>
<th>Encode</th>
<th>Decode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.37</td>
<td>0.43</td>
<td>3.80</td>
<td>3.57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4.49</td>
<td>0.77</td>
<td>5.26</td>
<td>6.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>10.48</td>
<td>1.84</td>
<td>12.32</td>
<td>16.18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>17.72</td>
<td>3.93</td>
<td>21.65</td>
<td>27.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>80.10</td>
<td>18.20</td>
<td>98.30</td>
<td>125.27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>163.09</td>
<td>36.47</td>
<td>199.56</td>
<td>259.71</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>336.37</td>
<td>73.52</td>
<td>409.89</td>
<td>549.62</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3275</td>
<td>608.74</td>
<td>130.03</td>
<td>738.77</td>
<td>941.60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>776.38</td>
<td>159.03</td>
<td>935.41</td>
<td>1206.64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>996.58</td>
<td>197.53</td>
<td>1194.11</td>
<td>1635.55</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### BasicVTItem

<table>
<thead>
<tr>
<th>Input Octet</th>
<th>Encode Input Time (msec)</th>
<th>Encode Output Time (msec)</th>
<th>Encoder Time (msec)</th>
<th>SerlDX Time (msec)</th>
<th>Encode Time (msec)</th>
<th>Decode Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.37</td>
<td>0.43</td>
<td>3.80</td>
<td>3.57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4.49</td>
<td>0.77</td>
<td>5.26</td>
<td>6.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>10.48</td>
<td>1.84</td>
<td>12.32</td>
<td>16.18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>17.72</td>
<td>3.93</td>
<td>21.65</td>
<td>27.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>80.10</td>
<td>18.20</td>
<td>98.30</td>
<td>125.27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>163.09</td>
<td>36.47</td>
<td>199.56</td>
<td>259.71</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>336.37</td>
<td>73.52</td>
<td>409.89</td>
<td>549.62</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3275</td>
<td>608.74</td>
<td>130.03</td>
<td>738.77</td>
<td>941.60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>776.38</td>
<td>159.03</td>
<td>935.41</td>
<td>1206.64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>996.58</td>
<td>197.53</td>
<td>1194.11</td>
<td>1635.55</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### PersonnelRecord

<table>
<thead>
<tr>
<th>Input Octet</th>
<th>Encode Input Time (msec)</th>
<th>Encode Output Time (msec)</th>
<th>Encoder Time (msec)</th>
<th>SerlDX Time (msec)</th>
<th>Encode Time (msec)</th>
<th>Decode Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td>4.30</td>
<td>1.10</td>
<td>5.40</td>
<td>4.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>475</td>
<td>5.71</td>
<td>2.00</td>
<td>7.71</td>
<td>5.85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1275</td>
<td>9.12</td>
<td>3.70</td>
<td>12.82</td>
<td>9.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3275</td>
<td>17.30</td>
<td>8.69</td>
<td>25.99</td>
<td>17.03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>23.95</td>
<td>12.35</td>
<td>36.30</td>
<td>23.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>44.79</td>
<td>23.95</td>
<td>68.74</td>
<td>43.22</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: The ED Library Performance
CHAPTER 6. PERFORMANCE AND APPLICATIONS

represent the BER octet sequence content length as the input data length increases.

<table>
<thead>
<tr>
<th>Input (octet)</th>
<th>1</th>
<th>127</th>
<th>128</th>
<th>255</th>
<th>256</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>3000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output (octet)</td>
<td>5</td>
<td>131</td>
<td>133</td>
<td>260</td>
<td>262</td>
<td>506</td>
<td>1006</td>
<td>2006</td>
<td>3006</td>
</tr>
<tr>
<td>Overhead (octet)</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 6.2: OCTET STRING Encode Octet Complexity

2. The octet-complexity for encoding a BasicVTPItem type is approximately linear as shown in Table 6.3 and Figure 6.2.

<table>
<thead>
<tr>
<th>Input (octet)</th>
<th>1</th>
<th>10</th>
<th>50</th>
<th>100</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>3275</th>
<th>4000</th>
<th>5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output (octet)</td>
<td>20</td>
<td>47</td>
<td>171</td>
<td>325</td>
<td>1525</td>
<td>3025</td>
<td>6025</td>
<td>9850</td>
<td>12025</td>
<td>15025</td>
</tr>
<tr>
<td>Overhead (octet)</td>
<td>19</td>
<td>37</td>
<td>121</td>
<td>225</td>
<td>1025</td>
<td>2025</td>
<td>4025</td>
<td>6575</td>
<td>8025</td>
<td>10025</td>
</tr>
</tbody>
</table>

Table 6.3: BasicVTPItem Encode Octet Complexity

3. The octet-complexity for encoding a PersonnelRecord is also approximately constant as shown in Table 6.4 and Figure 6.3. Notice that the minimum size of a PersonnelRecord data is 75.

<table>
<thead>
<tr>
<th>Input (octet)</th>
<th>75</th>
<th>187</th>
<th>197</th>
<th>199</th>
<th>314</th>
<th>327</th>
<th>500</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output (octet)</td>
<td>136</td>
<td>249</td>
<td>260</td>
<td>263</td>
<td>379</td>
<td>393</td>
<td>566</td>
<td>1066</td>
<td>5066</td>
<td>10066</td>
</tr>
<tr>
<td>Overhead (octet)</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
</tr>
</tbody>
</table>

Table 6.4: PersonnelRecord Encode Octet Complexity

Constant octet-complexity indicates that the extra octet overhead for carrying type information remains the same as the input value/size grows. Linear octet-complexity indicates that the extra octet overhead for carrying type information grows proportionately as the input value/size
Octet String Encode Octet Complexity

Input Data Length (octet)

Encode Octet Overhead (octet)

Figure 6.1: OCTET STRING Encode Octet Complexity
Figure 6.2: BASiCVTPITEM Encode Octet Complexity
Figure 6.3: PERSONNELRECORD Encode Octet Complexity
grows. It is important to note that the octet-complexity reflects the storage complexity of the BER encoding algorithms. All BER implementations must pay the same octet-complexity overhead. However, different BER implementations could have different time-complexity depending on their internal structures. Time performance profile data are also summarized in Table 6.1 and plotted in Figures 6.4, 6.5 and 6.62. We have the following observations:

1. Encoding and decoding times vary in the same trend, i.e. both increase as the input data-size increases.

2. For data of type OCTET STRING or PERSONNEL RECORD, the encoding time is greater than the decoding time. For data of type BASIC VTP ITEM the encoding time is less than the decoding time. This is because the BASIC VTP ITEM is a complicated CHOICE type and has many OPTIONAL sub-types. The decoder spends a considerable time in calling the ED library utility routine TestTag() to test tags in the input BER octet-sequence and to decide whether the values of the OPTIONAL sub-types should be decoded.

3. For encoding/decoding data-values with constant octet-complexity, the processing time seems to increase proportionately as the input data-size increases. This is shown in Figures 6.4 and 6.5. For encoding/decoding data-values with linear octet-complexity, the processing time seems to increase parabolically as the input data-size increases. This is shown in Figure 6.6. This may be because the input data-size is used to control loops in encoding and decoding routines

---

2 Encode time is the time spent on both Encode_() and serIDX().
Figure 6.4: OCTET STRING Encode/Decode Time Complexity
Figure 6.5: PERSONNELRECORD Encode/Decode Time Complexity
Figure 6.6: BASICVTPITEM Encode/Decode Time Complexity
CHAPTER 6. PERFORMANCE AND APPLICATIONS

6.2 Results of CASN1 Applications

Several applications have used CASN1. It has been used in the implementation of the ISO Virtual Terminal Protocol, in compiling the OSI X.500 Directory Protocol and the CCITT X.226 Presentation Protocol to generate the data type C definitions and their encoding/decoding routines. It is also used in a project integrating PDU ASN.1 definitions into Estelle specifications at the University of Montreal [18]. These applications demonstrate that CASN1 is a very useful tool for the automatic implementation of communication protocols.

6.2.1 The ISO Virtual Terminal Protocol

At the University of British Columbia, CASN1 has been used in an implementation of the ISO Virtual Terminal Protocol (VTP) defined in ISO/DIS 9041 [11]. The implementation is in C on UNIX 4.2 BSD.

The original VTP specification included 775 lines of ASN.1-definitions. Since no macro-definitions are used, it is directly fed into CASN1 without any modification. After compiling the VTP ASN.1 specifications, CASN1 generated 1,582 lines of C-definitions for the protocol defined data-types and 5,728 lines of C-code for the encoding/decoding routines for these data-types. The total source of the VTP implementation [17] includes 12,224 lines of C code. About 60% of it was automatically generated by CASN1.

6.2.2 The CCITT X.226 Presentation Protocol

CASN1 has been applied to the presentation layer protocol defined in the CCITT recommendation X.226 [3]. The 153 lines of the presentation layer protocol ASN.1 specification also do not contain any macro-definitions. CASN1 compiles the specification directly and generates 300 lines of C definitions for the data-types and 1,286 lines of C code for encoding/decoding.
routines for the data-types.

6.2.3 The OSI X.500 Directory Protocol

CASN1 has also been applied to the OSI X.500 Directory Protocol [12]. The protocol has 12 modules in ASN.1 specifications, totaling about 1,600 lines. They are stored in 12 files as input to CASN1. All macro-definitions were marked out as comments.

There are several syntax errors in the original specification. After several iterations of error correction and re-compilation, CASN1 generated about 1,100 lines of C-definitions for the data-types and 4,000 lines of C-code for encoding/decoding routines. This example shows that CASN1 can be used to help detect errors in ASN.1 protocol specifications.

6.2.4 The ASN.1/Estelle Integration Project

D. Ouimet and G. v. Bochmann at the University of Montreal have been working on a project to integrate ASN.1-definitions into Estelle-specifications [18]. They are building a software package that translates a protocol Estelle-specification into C and generates BER encoding and decoding routines for all defined data-types. Such a system may be built by integrating an ASN.1-C compiler with an Estelle-C compiler. After comparing and evaluating many existing packages, CASN1 has been chosen as their ASN.1-C compiler in the project for its high efficiency and user-friendly interface. The project is still in an early stage of development and more progress is expected.
Chapter 7

Conclusions and Future Research

This chapter summarizes the work of this thesis, points out the limitations in the current version of CASN1 and presents directions for future research.

7.1 Conclusions

The ED library is an efficient implementation of BER. It is flexible, easy to use and has proven helpful in many protocol implementation exercises. By structural analysis and comparison with other existing BER implementations, one can be easily convinced that the ED library provides potentially better timing performance than those referred to Section 1.3.

It has been recognized that to write data-type encoders and decoders manually is quite arduous, even with the help of the ED library routines. Therefore, CASN1 is designed and implemented as an automatic tool for translating protocol specifications from ASN.1 into C and for generating the encoders and decoders for every protocol-defined data-type. CASN1 is based on a subset of ASN.1 syntax rules because it does not handle ASN.1 macro-definitions. Users have to replace the macro-definitions with the operation-definition and error-definition before feeding the ASN.1 specification to CASN1. For cross-module type-references, some manual involvement may still be necessary. Therefore, a user should not fully depend on the output.
generated by the current version of CASN1. Even with these deficiencies, one can still be easily convinced by many application examples that CASN1 is a very powerful tool for speeding up protocol implementations.

7.2 Future Work

The current version of CASN1 could be further improved in the following aspects. First, the structure of CASN1 could be re-organized so that more functionality can be added to improve the user interface. In the current version, CASN1 translates a protocol module by module without keeping information among the modules. This results in two potential problems:

1. The current version of CASN1 can not handle cross-module references. Manual involvement may be necessary if maintaining inter-module relationships is required.

2. As discussed in Chapter 5, CASN1 assumes the top-down order of type-definitions in the input ASN.1 specifications, for the purpose of simplifying the compiler implementation. This could be a potential problem when cross-module reference must be maintained, because modules are not necessarily ordered according to type reference relationship.

To improve CASN1, all information about type-definitions, value-definitions, operation-definitions as well as error-definitions can be saved in an intermediate data structure, called the definition table, during parsing. With the global information recorded in the definition table, two useful facilities could be built:

1. A module-configuration mechanism to deal with cross-module relationships, e.g., IMPORTing or EXPORTing type-definitions and value-definitions among ASN.1 modules; using MODULE.TYPE to reference external types and MODULE.VALUE to reference external values etc.
CHAPTER 7. CONCLUSIONS AND FUTURE RESEARCH

2. A topological sorting mechanism to order the type-definitions according to type reference relationships.

Second, it would be very useful, and not very difficult, to port the CASN1 implementation to an ASN.1-Pascal or an ASN.1-C++ compiler.

Third, a complete ASN.1 compiler must have a parser that dynamically extend its syntax rules because ASN.1 syntax rules are very flexible and extensible. One possible solution is to build a knowledge-based compiler with self-learning capability. It learns new knowledge of parsing (i.e., new syntax rules) dynamically during parsing ASN.1 macro-definitions and detects infinite reduction using the knowledge it has learned. The testing of infinite reduction might be mapped to a problem of detecting loops in a directed graph represents the rules of syntax reduction. All in all, this problem remains an interesting research topic for knowledge-based compiler design and implementation.
Bibliography


Appendix A

BNF of ASN.1 Syntax Rules

ModuleDefinition ::= modulereference
                  DEFINITIONS "::="
                  BEGIN ModuleBody END

ModuleBody ::= AssignmentList|Empty

AssignmentList ::= Assignment
                 | AssignmentList Assignment

Assignment ::= Typeassignment|Valueassignment

Externaltypereference ::= modulereference Dot typereference

Externalvaluereference ::= modulereference Dot valuereference

DefinedType ::= Externaltypereference
              | typereference

DefinedValue ::= Externalvaluereference
               | valuereference

Typeassignment ::= typereference "::=" Type

Valueassignment ::= valuereference "::=" Value

Type ::= BuiltinType|DefinedType
APPENDIX A. BNF OF ASN.1 SYNTAX RULES

| BuiltinType        | ::= BooleanType    |
|                   |                  |
|                   | | IntegerType      |
|                   | | BitStringType    |
|                   | | OctetStringType  |
|                   | | NullType         |
|                   | | SequenceType     |
|                   | | SequenceOfType   |
|                   | | setType          |
|                   | | SetOfType        |
|                   | | ChoiceType       |
|                   | | SelectionType    |
|                   | | TaggedType       |
|                   | | AnyType          |
|                   | | ExternalType     |
|                   | | CharacterSetType |
|                   | | UsefulType       |

| NameType          | ::= identifier  Type |
|                   | | Type              |
|                   | | SelectionType     |

| Value             | ::= BuiltinValue |
|                   | | DefinedValue     |

| BuiltinValue      | ::= BooleanValue |
|                   | | IntegerValue     |
|                   | | BitStringValue   |
|                   | | OctetStringValue |
|                   | | NullValue        |
|                   | | SequenceValue    |
|                   | | SequenceOfValue  |
|                   | | SetValue         |
|                   | | SetOfType        |
|                   | | ChoiceValue      |
|                   | | SelectionValue   |
|                   | | TaggedValue      |
|                   | | AnyValue         |
|                   | | CharacterSetValue |

| NamedValue        | ::= identifier  Value |
|                   | | Value             |
APPENDIX A. BNF OF ASN.1 SYNTAX RULES

BooleanType ::= BOOLEAN

BooleanValue ::= TRUE|FALSE

IntegerType ::= INTEGER
  | INTEGER "{" NamedNumberList "}"  

NamedNumberList ::= NamedNumber
  | NamedNumberList "," NamedNumber
  | Number

NamedNumber ::= identifier "(" SignedNumber ")"
  | identifier "(" DefinedValue ")"

SignedNumber ::= number
  | "-" number

IntegerValue ::= SignedNumber
  | identifier

BitStringType ::= BITSTRING
  | BITSTRING "{" NamedBitList "}"  

NamedBitList ::= NamedBit
  | NamedBitList "," NamedBit

NamedBit ::= identifier "(" number ")"
  | identifier "(" DefinedValue ")"

BitStringValue ::= bstring
  | hstring
  | 
    
      
        
          "{" IdentifierList "}"

      
    
        "{" "}"

IdentifierList ::= identifier
  | IdentifierList "," identifier

OctetStringType ::= OCTETSTRING

OctetStringValue ::= bstring
APPENDIX A. BNF OF ASN.1 SYNTAX RULES

NullType ::= NULL
NullValue ::= NULL

SequenceType ::= SEQUENCE "{" ElementTypeList "}" 

ElementTypeList ::= ElementType 
                  | ElementTypeList "," ElementType

ElementType ::= NamedType 
              | NamedType OPTIONAL 
              | NamedType DEFAULT Value 
              | COMPONENTSOF Type

SequenceValue ::= "{" ElementValueList "}" 
                | "{""}"

ElementValueList ::= NamedValue 
                  | ElementValueList "," NamedValue

SequenceOfType ::= SEQUENCEOF Type

SequenceOfValue ::= "{""ValueList""}" 
                 | "{""}"

ValueList ::= Value 
           | ValueList "," Value

SetType ::= SET "{" ElementList "}"

SetValue ::= "{""ElementValueList""}" 
            | "{""}"

SetOfType ::= SETOF Type

SetOfValue ::= "{"" ValueList ""}" 
             | "{""}"

ChoiceType ::= CHOICE "{" AlternativeList "}"
APPENDIX A. BNF OF ASN.1 SYNTAX RULES

AlternativeList ::= NamedType
                | AlternativeList "," NamedType
ChoiceValue ::= NamedValue
SelectionType ::= identifier < Type
SelectionValue ::= NamedValue
TaggedType ::= Tag Type
             | Tag IMPLICIT Type
Tag ::= "[" Class ClassNumber "]"
ClassNumber ::= number
             | DefinedValue
Class ::= UNIVERSAL
       | APPLICATION
       | PRIVATE
       | Empty
TaggedValue ::= Value
AnyType ::= ANY
AnyValue ::= Type Value
CharacterSetType ::= typereference
CharacterSetValue ::= cstring
UsefulType ::= typereference
EnumeratedType ::= ENUMERATED "{" Enumeration "}"
Enumeration ::= NamedNumber
               | NamedNumber "," Enumeration
RealType ::= REAL
RealValue ::= NumericRealValue | SpecialRealValue

NumericRealValue ::= "{" Mantissa "," Base "," Exponent "}" | number

Mantissa ::= SignedNumber

Base ::= number

Exponent ::= SignedNumber

SpecialRealValue ::= PINFINITY | MINFINITY

EncryptedType ::= ENCRYPTED Type

SubType ::= ParentType SubtypeSpec | SET SizeConstraint Of Type | SEQUENCE SizeConstraint Of Type

ParentType ::= Type

SubtypeSpec ::= "(" SubtypeAlternative SubtypeAlternativeList ")"

SubtypeAlternativeList ::= empty | "|" SubtypeAlternative SubtypeAlternativeList

SubtypeAlternative ::= SubtypeValueSet | SubtypeConstraint

SubtypeValueSet ::= Value | ContainedSubtype | ValueRange | PermittedAlphabet
APPENDIX A. BNF OF ASN.1 SYNTAX RULES

SubtypeConstraint ::= SizeConstraint
                   | InnerTypeConstraints

ContainedSubtype ::= INCLUDES Type

ValueRange ::= LowerEndpoint ".." UpperEndpoint

LowerEndpoint ::= LowerEndValue
                | LowerEndValue Larrow

UpperEndpoint ::= UpperEndValue
                | Larrow UpperEndValue

LowerEndValue ::= Value
                | MIN

UpperEndValue ::= Value
                | MAX

SizeConstraint ::= SIZE SubtypeSpec

PermittedAlphabet ::= FROM SubtypeSpec

InnerTypeConstraints ::= WITH COMPONENT
                        SingleTypeConstraint
                        | WITH COMPONENTS
                        MultipleTypeConstraints

SingleTypeConstraint ::= SubtypeSpec

MultipleTypeConstraints ::= FullSpecification
                           | PartialSpecification

FullSpecification ::= "{" TypeConstraints "}"n

PartialSpecification ::= "{" "..." "," TypeConstraints "}"n

TypeConstraints ::= NamedConstraint
                   | NamedConstraint ","
                   TypeConstraints
APPENDIX A. BNF OF ASN.1 SYNTAX RULES

NamedConstraint ::= identifier Constraint
                 | Constraint

Constraint ::= ValueConstraint PresenceConstraint

ValueConstraint ::= empty
                  | SubtypeSpec

PresenceConstraint ::= empty
                    | PRESENT
                    | ABSENT
Appendix B

BNF of CASN1 Syntax Rules

ModuleDefinition ::= ModuleIdentifier
  DEFINITIONS TagDefault "::="
  BEGIN ModuleBody END

ModuleIdentifier ::= typereference
  AssignedIdentifier

TagDefault ::= empty
  | EXPLICIT TAGS
  | IMPLICIT TAGS

AssignedIdentifier ::= empty
  | identifier
  | ObjectIdentifierValue

ModuleBody ::= Exports Imports AssignmentList

Exports ::= empty
  | EXPORTS SymbolsExported "::"

SymbolsExported ::= empty
  | SymbolList

Imports ::= empty
  | IMPORTS
  SymbolsImported "::"
APPENDIX B. BNF OF CASN1 SYNTAX RULES

SymbolsImported ::= empty
| SymbolsFromModuleList

SymbolsFromModuleList ::= SymbolsFromModule
| SymbolsFromModule SymbolsFromModuleList

SymbolsFromModule ::= SymbolList FROM ModuleIdentifier

SymbolList ::= Symbol
| Symbol "," SymbolList

Symbol ::= typereference
| identifier

AssignmentList ::= empty
| Assignment AssignmentList

Assignment ::= Typeassignment
| Valueassignment
| OperationDefinition
| ErrorDefinition

Externaltypereference ::= modulereference
| modulereference
| typereference

Externalvaluereference ::= modulereference
| modulereference
| identifier

DefinedValue ::= Externalvaluereference
| identifier

Typeassignment ::= typereference ":=" Type

Valueassignment ::= identifier TypeAssignValue

TypeAssignValue ::= BooleanType ":=" BooleanValue
| IntegerType ":=" IntegerValue
| BitStringType ":=" BitStringValue
| OctetStringType ":=" OctetStringValue
APPENDIX B. BNF OF CASN1 SYNTAX RULES

NullType ::= Null

CharacterStringType ::= CharacterStringValue

ObjectIdentifierType ::= ObjectIdentifierValue

OBJDSCRPT ::= CharacterStringValue

UTCTime ::= CharacterStringValue

GNLTime ::= CharacterStringValue

RealType ::= RealValue

EXTERNAL ::= SequenceValue

EncryptedType ::= BitStringValue

DefinedType ::= Value

ChoiceType ::= Value

SequenceOfType ::= SequenceOfValue

SetOfType ::= SetOfValue

SelectionType ::= Value

TaggedType ::= Value

AnyType ::= AnyValue

EnumeratedType ::= identifier

Type ::= BuiltinType

BuiltInType ::=

BooleanType

IntegerType

BitStringType

OctetStringType

NullType

SequenceType

SequenceOfType

SetType

SetOfType

ChoiceType

SelectionType

TaggedType

AnyType

ObjectIdentifierType

CharacterStringType

UsefulType

EnumeratedType

RealType

EncryptedType
APPENDIX B. BNF OF CASN1 SYNTAX RULES

DefinedType ::= Externaltypereference
               | typereference
               | SubType

NamedType ::= identifier Type
            | Type

Value ::= True
        | False
        | SignedNumber
        | CharacterStringValue
        | Null
        | "{" ValueList "}"
        | "{" "}".
        | ObjectIdentifierValue
        | RealValue
        | AnyValue
        | identifier Value
        | DefinedValue

ValueList ::= Value
             | ValueList "," Value

BooleanType ::= BOOLEAN

BooleanValue ::= True
               | False

IntegerType ::= INTEGER Intrail

Intrail ::= empty
         | "{" NamedNumberList "}".

NamedNumberList ::= NamedNumber
                  | NamedNumberList "," NamedNumber

NamedNumber ::= identifier "(" NumberList ")"

NumberList ::= SignedNumber
              | DefinedValue
APPENDIX B. BNF OF CASN1 SYNTAX RULES

SignedNumber ::= number
   | Minus number

IntegerValue ::= SignedNumber
   | identifier

BitStringType ::= BITSTRING Bitrail

Bitrail ::= empty
   | "{" NamedBitList "}"n

NamedBitList ::= NamedBit
   | NamedBitList "," NamedBit

NamedBit ::= identifier "(" NumberForm ")"

NumberForm ::= number
   | DefinedValue

BitStringValue ::= bstring
   | hstring
   | "{" IdentifierList "}"n
   | "{" "}"

IdentifierList ::= identifier
   | IdentifierList "," identifier

OctetStringType ::= OCTET STRING

OctetStringValue ::= bstring
   | hstring

NullType ::= NULL

SequenceType ::= SEQUENCE 
   | "{" ElementTypeList "}"n

ElementTypeList ::= empty
   | ElementType
   | ElementTypeList "," ElementType

ElementType ::= NameType
APPENDIX B. BNF OF CASN1 SYNTAX RULES

```
| OPTIONAL NamedType |
| NamedType DEFAULT Value |
| COMPONENTS Of typereference |

SequenceValue ::= "{" ValueList "}"
| "{" "}"

SequenceOfType ::= SEQUENCE Of Type
| SEQUENCE

SequenceOfValue ::= "{" ValueList "}"
| "{" "}"

SetType ::= SET "{" ElementTypeList "}"

SetValue ::= "{" ValueList "}"
| "{" "}"

SetOfType ::= SET Of Type
| SET

SetOfValue ::= "{" ValueList "}"

ChoiceType ::= CHOICE "{" AlternativeTypeList "}"

AlternativeTypeList ::= NamedType
| AlternativeTypeList "," NamedType

SelectionType ::= identifier Larrow typereference

TaggedType ::= Tag Type
| Tag IMP Type
| Tag EXPLICIT Type

Tag ::= "[" Class ClassNumber "]"

ClassNumber ::= number
| DefinedValue

Class ::= empty
| UNIVERSAL
```
APPENDIX B. BNF OF CASN1 SYNTAX RULES

AnyType ::= ANY
          | ANY DEFINED BY identifier

AnyValue ::= Type Value

ObjectIdentifierType ::= OBJECTID

ObjectIdentifierValue ::= "{" ObjIdComponentList "}"
          | "{" DefinedValue
          | "{" ObjIdComponentList "}"

ObjIdComponentList ::= ObjIdComponent
                      | ObjIdComponent ObjIdComponentList

ObjIdComponent ::= NumberForm
                 | NameAndNumberForm

NameAndNumberForm ::= identifier "(" NumberForm ")"

CharacterStringType ::= Nums
                       | PrintableString
                       | TelexString
                       | VideotexString
                       | IA5String
                       | GraphicString
                       | VisibleString

UsefulType ::= UTCTime
              | GNLTTime
              | EXTERNAL
              | OBJECT DISCRIPTOR

EnumeratedType ::= ENUMERATED
             | "{" Enumeration "}"

Enumeration ::= NamedNumber
              | NamedNumber "," Enumeration
APPENDIX B. BNF OF CASN1 SYNTAX RULES

RealType ::= REAL

RealValue ::= NumericRealValue |
            SpecialRealValue

NumericRealValue ::= "{" Mantissa "," |
                   number |
                   Base "," |
                   Exponent "}"

Mantissa ::= SignedNumber

Base ::= number

Exponent ::= SignedNumber

SpecialRealValue ::= PINFINITY |
                   MINFINITY

EncryptedType ::= ENCRYPTED Type

SubType ::= ParentType SubtypeSpec |
           SET SizeConstraint Of Type |
           SEQUENCE SizeConstraint Of Type

ParentType ::= Type

SubtypeSpec ::= "{" SubtypeAlternative |
               SubtypeAlternativeList "}"

SubtypeAlternativeList ::= empty |
                         "|" SubtypeAlternative |
                         SubtypeAlternativeList

SubtypeAlternative ::= SubtypeValueSet |
                     SubtypeConstraint

SubtypeValueSet ::= Value |
                   ContainedSubtype |
                   ValueRange |
                   PermittedAlphabet
APPENDIX B. BNF OF CASN1 SYNTAX RULES

SubtypeConstraint ::= SizeConstraint
| InnerTypeConstraints

ContainedSubtype ::= INCLUDES Type

ValueRange ::= LowerEndpoint ".." UpperEndpoint

LowerEndpoint ::= LowerEndValue
| LowerEndValue Larrow

UpperEndpoint ::= UpperEndValue
| Larrow UpperEndValue

LowerEndValue ::= Value
| MIN

UpperEndValue ::= Value
| MAX

SizeConstraint ::= SIZE SubtypeSpec

PermittedAlphabet ::= FROM SubtypeSpec

InnerTypeConstraints ::= WITH COMPONENT
SingleTypeConstraint
| WITH COMPONENTS
MultipleTypeConstraints

SingleTypeConstraint ::= SubtypeSpec

MultipleTypeConstraints ::= FullSpecification
| PartialSpecification

FullSpecification ::= "{" TypeConstraints "}%"

PartialSpecification ::= "{" ".." ","
| TypeConstraints "}%"

TypeConstraints ::= NamedConstraint
| NamedConstraint "]"
APPENDIX B. BNF OF CASN1 SYNTAX RULES

TypeConstraints

NamedConstraint ::= identifier Constraint
                 | Constraint

Constraint ::= ValueConstraint
            PresenceConstraint

ValueConstraint ::= empty
                 | SubtypeSpec

PresenceConstraint ::= empty
                    | PRESENT
                    | ABSENT

OperationDefinition ::= identifier OPERATION
                      OperationTypeNotation
                      ::= number

OperationTypeNotation ::= empty
                        | ARGUMENT typereference NamedType
                        Result
                        Errors

Result ::= empty
         | RESULT typereference NamedType

Errors ::= empty
        | ERRORS "{" ErrorNames "}""

ErrorNames ::= empty
            | identifier
            | identifier "," ErrorNames

ErrorDefinition ::= identifier ERROR
                  ErrorTypeNotation
                  ::= number

ErrorTypeNotation ::= empty
                    | PARAMETER typereference NamedType
Appendix C

Example of Structured Type Encoding and Decoding

ASN.1 Definition:

```
-- This is an example of ASN.1 structured type
STRUCT Definitions ::= BEGIN
Var1 ::= SET { r00 INTEGER OPTIONAL ,
              r01 Var1 ,
              r02 SEQUENCE { r10 ENUMERATED
                            { id1(10), id2(20), id3(30) } ,
                            r11 SET OF Var2 ,
                            r12 NULL OPTIONAL }}

Var2 ::= BOOLEAN
END

C Definition:

typedef bool Var2; /* BOOLEAN */

typedef enum T22 { /* ENUMERATED Type */
  id1 = 10 , id2 = 20 , id3 = 30 } T22;

typedef struct T11 { /* SET/SEQ */
  T22 r10; /* T22 */
  LIST OF(Var2) r11; /* Var2 */
  int *r12; /* NULL(optional) */
} T11;
```
typedef struct Var1 {
    int *r00;       /* SET/SEQ */
    int *r01;       /* INTEGER(optional) */
    struct Var1 *r02; /* Var1(recursive) */
} Var1;

Encoders:

#include "STRUCT.defs.c"

IDX *
Encode_Var2(class, form, code, Var2_VP)
    short class;
    short form;
    long code;
    Var2 *Var2_VP;
{
    IDX *p;
    p = (IDX *) Encode_bool(class, form, code, Var2_VP);
    return (p);
}

IDX *
Encode_T22(class, form, code, T22_VP)
    short class;
    short form;
    long code;
    T22 *T22_VP;
{
    IDX *p;
    p = (IDX *) Encode_int(class, form, code, T22_VP);
    return (p);
}

IDX *
Encode_T11(class, form, code, T11_VP)
    short class;
    short form;
    long code;
APPENDIX C. EXAMPLE OF STRUCTURED TYPE ENCODING AND DECODING

T11 *T11_VP;
{
    IDX *p;
    p = (IDX *) Encode_struct_beg(class, form, code);
    Encode_T22(UNIVERSAL, PRIMITIVE, 10, &(T11_VP->r10));
    Encode_struct_beg(UNIVERSAL, CONSTRUCTOR, CODE_SET, &(T11_VP->r11));
    ListFor(T11_VP->r11)
        Encode_Var2(UNIVERSAL, PRIMITIVE, 1, (T11_VP->r11)->next->item);
    Encode_struct_end(DEFINITE);
    if (T11_VP->r12 != NULL)
        Encode_null(UNIVERSAL, PRIMITIVE, CODE_NULL, T11_VP->r12);
    Encode_struct_end(DEFINITE);
}

return (p);

IDX *
Encode_Var1(class, form, code, Var1_VP)
    short class;
    short form;
    long code;
    Var1 *Var1_VP;
{
    IDX *p;
    p = (IDX *) Encode_struct_beg(class, form, code);
    if (Var1_VP->r00 != NULL)
        Encode_int(UNIVERSAL, PRIMITIVE, CODE_INTEGER, Var1_VP->r00);
    Encode_Var1(UNIVERSAL, CONSTRUCTOR, 17, Var1_VP->r01);
    Encode_T11(UNIVERSAL, CONSTRUCTOR, 16, &(Var1_VP->r02));
APPENDIX C. EXAMPLE OF STRUCTURED TYPE ENCODING AND DECODING 103

```c
Encode_struct_end(DEFINITE);

    return (p);
}

Decoders:

#include "STRUCT.defs.c"

Var2 *
Decode_Var2(class, form, code, B, Var2_VP)
    short    class;
    short    form;
    long     code;
    IOP      *B;
    bool     *Var2_VP;
{
    Var2_VP = (bool *) Decode.bool (class, form, code, B, Var2_VP);

    return (Var2_VP);
}

T22 *
Decode_T22(class, form, code, B, T22_VP)
    short    class;
    short    form;
    long     code;
    IOP      *B;
    T22      *T22_VP;
{
    T22_VP = (T22 *) Decode.int (class, form, code, B, T22_VP);

    return (T22_VP);
}

T11 *
Decode_T11(class, form, code, B, T11_VP)
    short    class;
    short    form;
    long     code;
    IOP      *B;
    T11      *T11_VP;
```
APPENDIX C. EXAMPLE OF STRUCTURED TYPE ENCODING AND DECODING 104

```c
{
    if (T11_VP == NULL)
        T11_VP = (T11 *) GETBUF(sizeof(T11));

    Decode_struct_beg(class, form, code, B);

    Decode_T22(UNIVERSAL, PRIMITIVE, 10, B, &(T11_VP->r10));

    Decode_struct_beg(UNIVERSAL, CONSTRUCTOR, CODE_SET, B, &(T11_VP->r11));
    while (ListEnd(B))
        ListAppend(&(T11_VP->r11), Decode_Var2(UNIVERSAL, PRIMITIVE, 1, B, 0));
    Decode_struct_end(B);

    if (TestTag(UNIVERSAL, PRIMITIVE, CODE_NULL, B) != FALSE)
        T11_VP->r12 = (int *) Decode.null(UNIVERSAL, PRIMITIVE, CODE_NULL, B, T11_VP->r12);
    Decode_struct_end(B);

    return (T11_VP);
}

Var1 *
Decode_Var1(class, form, code, B, Var1_VP)
    short    class;
    short    form;
    long     code;
    IODP     *B;
    Var1     *Var1_VP;
{
    if (Var1_VP == NULL)
        Var1_VP = (Var1 *) GETBUF(sizeof(Var1));

    Decode_struct_beg(class, form, code, B);

    if (TestTag(UNIVERSAL, PRIMITIVE, CODE_INTEGER, B) != FALSE)
        Var1_VP->r00 = (int *) Decode_int (UNIVERSAL, PRIMITIVE, CODE_INTEGER, B, Var1_VP->r00);

    Var1_VP->r01 = (struct Var1 *) Decode_Var1
```
( UNIVERSAL, CONSTRUCTOR, 17, B, Var1_VP->r01);

Decode_T11(UNIVERSAL, CONSTRUCTOR, 16, B, &(Var1_VP->r02));

Decode_struct_end(B);

return (Var1_VP);
}
Appendix D

Example of CHOICE Type Encoding and Decoding

ASN.1 Definition

D := CHOICE{
    d [0] NULL,
    e [1] PrintableString
}
C := CHOICE{
    f [2] NULL,
    g [3] PrintableString
}
A := CHOICE{
    b D,
    c C
}

C Definition

typedef struct D {
    int choice;
    union {
        int d;
        OCTS e;
    }
    data
} D;

typedef struct C {
    int choice;
    union {
        int f;
        OCTS g;
    }
} C;
APPENDIX D. EXAMPLE OF CHOICE TYPE ENCODING AND DECODING

#define C_g_$tag 0xa0000003

} data
} C;

typedef struct A {
    int choice; /* indicate the choice of data */
    union {
        D b; /* D */
        C c; /* C */
    }
    data
} A;

Encoders:

IDX *
Encode_D(class, form, code, D_VP)
{
    short class;
    short form;
    long code;
    D *D_VP;

    IDX *p;

    switch (D_VP->choice) {
    case D_d_$tag:
        p = (IDX *) Encode_struct_beg(CONTEXT, CONSTRUCTOR, 0);
        Encode_null(UNIVERSAL, PRIMITIVE, CODE_NULL, &(D_VP->data.d));
        Encode_struct_end(DEFINITE);
        break;
    case D_e_$tag:
        p = (IDX *) Encode_struct_beg(CONTEXT, CONSTRUCTOR, 1);
        Encode_octs(UNIVERSAL, PRIMITIVE, CODE_PRINTABLE_STRING,
                     &(D_VP->data.e));
        Encode_struct_end(DEFINITE);
        break;
    default:
        fprintf(stderr, "Encode_D : No such choice\n");
        exit();
    }

    return (p);
}
APPENDIX D. EXAMPLE OF CHOICE TYPE ENCODING AND DECODING

IDX *
Encode_C(class, form, code, C_VP)
    short    class;
    short    form;
    long     code;
    C        *C_VP;
{
    IDX     *p;

    switch (C_VP->choice) {
    case C_f_$tag:
        p = (IDX *) Encode_struct_beg(CONTEXT, CONSTRUCTOR, 2);
        Encode.null(UNIVERSAL, PRIMITIVE, CODE.NULL, &(C_VP->data.f));
        Encode_struct_end(DEFINITE);
        break;
    case C_g_$tag:
        p = (IDX *) Encode_struct_beg(CONTEXT, CONSTRUCTOR, 3);
        Encode.octs(UNIVERSAL, PRIMITIVE, CODE_PRINTABLE_STRING,
                     ft(C_VP->data.g));
        Encode_struct_end(DEFINITE);
        break;
    default:
        fprintf(stderr, "Encode_C : No such choice\n");
        exitO ;
    }

    return (p);
}

IDX *
Encode_A(class, form, code, A_VP)
    short    class;
    short    form;
    long     code;
    A        *A_VP;
{
    IDX     *p;

    switch (A_VP->choice) {
    case D_d_$tag:
APPENDIX D. EXAMPLE OF CHOICE TYPE ENCODING AND DECODING

case D_e_$tag:
    p = (IDX *) Encode_D(-1, -1, -1, &(A_VP->data.b));
    break;

Decoders:

D *
Decode_D(class, form, code, B, D_VP)
short class;
short form;
long code;
IOP *B;
D *D_VP;
{
    if (D_VP == NULL)
        D_VP = (D *) GETBUF(sizeof(D));

    D_VP->choice = GetTag(B);
    switch (D_VP->choice) {
    case D_d_$tag:
        Decode_struct_beg(CONTEXT, CONSTRUCTOR, 0, B);
        Decode.null(UNIVERSAL, PRIMITIVE, CODE.NULL, B,
            &(D_VP->data.d));
        Decode_struct_end(B);
        break;
    case D_e_$tag:
        Decode_struct_beg(CONTEXT, CONSTRUCTOR, 1, B);
        Decode.octs(UNIVERSAL, PRIMITIVE, CODE_PRINTABLE_STRING, B,
            &(D_VP->data.e));
        Decode_struct_end(B);
        break;
default:
    fprintf(stderr, "Decode_D : No such choice\n");
    exit(0);
}

return (D_VP);
}

C *
Decode_C(class, form, code, B, C_VP)
short class;
short form;
long code;
IOP *B;
C *C_VP;
{
    if (C_VP == NULL)
        C_VP = (C *) GETBUF(sizeof(C));

    C_VP->choice = GetTag(B);
    switch (C_VP->choice) {
    case C_f_$tag:
        Decode_struct_beg(CONTEXT, CONSTRUCTOR, 2, B);
        Decode.null(UNIVERSAL, PRIMITIVE, CODE.NULL, B,
                     &(C_VP->data.f));
        Decode_struct_end(B);
        break;
    case C_g_$tag:
        Decode_struct_beg(CONTEXT, CONSTRUCTOR, 3, B);
        Decode.octs(UNIVERSAL, PRIMITIVE, CODE_PRINTABLE_STRING, B,
                     &(C_VP->data.g));
        Decode_struct_end(B);
        break;
    default:
        fprintf(stderr, "Decode_C : No such choice\n");
        exit(0);
    }

    return (C_VP);
}
APPENDIX D. EXAMPLE OF CHOICE TYPE ENCODING AND DECODING

A *
Decode_A(class, form, code, B, A_VP)
  short class;
  short form;
  long code;
  IOP *B;
  A *A_VP;
{
  if (A_VP == NULL)
      A_VP = (A *) GETBUF(sizeof(A));

  A_VP->choice = GetTag(B);
  switch (A_VP->choice) {
  case D_d_$tag:
      Decode_D(-1, -1, -1, B, &(A_VP->data.b));
      break;
  case C_f_$tag:
  case C_g_$tag:
      Decode_C(-1, -1, -1, B, &(A_VP->data.c));
      break;
  default:
      fprintf(stderr, "Decode_A : No such choice\n");
      exit();
  }

  return (A_VP);
}
Appendix E

Example of Using CASN1 and ED Library

1. The ASN.1 Definition (Input of CASN1) ¹

```
PERSONNELRECORD DEFINITIONS ::= 
BEGIN

PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET {
    Name ,
    [0] VisibleString ,
    EmployeeNumber ,
    [1] Date ,
    [2] Name ,
    [3] IMPLICIT
    SEQUENCE OF
    ChildInformation
    DEFAULT {} }

ChildInformation ::= SET {
    Name ,
    dateOfBirth [0] Date }

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE
```

¹This is the example used by both CCITT X.409 and ISO ASN.1/BER.
APPENDIX E. EXAMPLE OF USING CASN1 AND ED LIBRARY

{  
givenName VisibleString,  
initial VisibleString,  
familyName VisibleString}  

EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER  

Date ::= [APPLICATION 3] IMPLICIT VisibleString  

END  

2. The C Definition (Output of CASN1)  

File PERSONNELRECORD.defs.c:

#include "/spring/yang/pub/defs.h"  
#include "/spring/yang/pub/element.h"  

typedef OCTS Date;  /* APPLICATION 3 VisibleString */  

IDX *Encode_Date();  
Date *Decode_Date();  

typedef int EmployeeNumber;  /* APPLICATION 2 INTEGER */  

IDX *Encode_EmployeeNumber();  
EmployeeNumber *Decode_EmployeeNumber();  

typedef struct Name {  /* APPLICATION 1 SET/SEQ */  
  OCTS givenName;  /* VisibleString */  
  OCTS initial;  /* VisibleString */  
  OCTS familyName;  /* VisibleString */  
} Name;  

IDX *Encode_Name();  
Name *Decode_Name();  

typedef struct  
ChildInformation {  /* SET/SEQ */  
  Name r11;  /* Name */  
  Date dateOfBirth;  /* CONTEXT 0 Date */  
} ChildInformation;
APPENDIX E. EXAMPLE OF USING CASNI AND ED LIBRARY

IDX *Encode_ChildInformation();
ChildInformation *Decode_ChildInformation();

typedef struct
PersonnelRecord {
    /* APPLICATION 0 SET/SEQ */
    Name rll; /* Name */
    OCTS title; /* CONTEXT 0 VisibleString */
    EmployeeNumber number; /* EmployeeNumber */
    Date dateOfHire; /* CONTEXT 1 Date */
    Name nameOfSpouse; /* CONTEXT 2 Name */
    LIST OF(ChildInformation) child;
    /* CONTEXT 3 ChildInformation */
} PersonnelRecord;

IDX *Encode_PersonnelRecord();
PersonnelRecord *Decode_PersonnelRecord();

File PERSONNELRECORD.encode.c:

#include "PERSONNELRECORD.defs.c"

IDX *
Encode_Date(class, form, code, Date_VP)
short class;
short form;
long code;
Date *Date_VP;
{
    IDX *p;

    p = (IDX *) Encode_octs(class, form, code, Date_VP);

    return (p);
}

IDX *
Encode_EmployeeNumber(class, form, code, EmployeeNumber_VP)
short class;
short form;
long code;
EmployeeNumber *EmployeeNumber_VP;
{
IDX *p;

p = (IDX *) Encode_int(class, form, code, EmployeeNumber_VP);

return (p);

IDX * Encode_Name(class, form, code, Name_VP)
short class;
short form;
long code;
Name *Name_VP;
{
IDX *p;

p = (IDX *) Encode_struct_beg(class, form, code);

Encode_octs(UNIVERSAL, PRIMITIVE, CODE_VSB_STRING, &Name_VP->givenName);
Encode_octs(UNIVERSAL, PRIMITIVE, CODE_VSB_STRING, &Name_VP->initial);
Encode_octs(UNIVERSAL, PRIMITIVE, CODE_VSB_STRING, &Name_VP->familyName);

Encode_struct_end(DEFINITE);

return (p);
}

IDX * Encode_Childlnformation(class, form, code, Childlnformation_VP)
short class;
short form;
long code;
Childlnformation *Childlnformation_VP;
{
IDX *p;

p = (IDX *) Encode_struct_beg(class, form, code);
APPENDIX E. EXAMPLE OF USING CASE AND ED LIBRARY

```c
Encode_Name(APPLICATION, CONSTRUCTOR, 1,
            &(ChildInformation_VP->rll));

Encode_struct_beg(CONTEXT, CONSTRUCTOR, 0);
Encode_Date(APPLICATION, PRIMITIVE, 3,
            &(ChildInformation_VP->dateOfBirth));
Encode_struct_end(DEFINITE);

return (p);
```

```c
IDX *
Encode_PersonnelRecord(class, form, code, PersonnelRecord_VP)
{
  short    class;
  short    form;
  long     code;
  PersonnelRecord *PersonnelRecord_VP;

  IDX *p;
  p = (IDX *) Encode_struct_beg(class, form, code);

  Encode_Name(APPLICATION, CONSTRUCTOR, 1,
              &(PersonnelRecord_VP->rll));

  Encode_struct_beg(CONTEXT, CONSTRUCTOR, 0);
  Encode_octs(UNIVERSAL, PRIMITIVE, CODE_VSB_STRING,
               &(PersonnelRecord_VP->title));
  Encode_struct_end(DEFINITE);

  Encode_EmployeeNumber(APPLICATION, PRIMITIVE, 2,
                         &(PersonnelRecord_VP->number));

  Encode_struct_beg(CONTEXT, CONSTRUCTOR, 1);
  Encode_Date(APPLICATION, PRIMITIVE, 3,
              &(PersonnelRecord_VP->dateOfHire));
  Encode_struct_end(DEFINITE);

  Encode_struct_beg(CONTEXT, CONSTRUCTOR, 2);
  Encode_Name(APPLICATION, CONSTRUCTOR, 1,
```
APPENDIX E.  EXAMPLE OF USING CASN1 AND ED LIBRARY

&(PersonnelRecord_VP->nameOfSpouse));
Encode_struct_end(DEFINITE);

Encode_struct_beg(CONTEXT, CONSTRUCTOR, 3,
    &(PersonnelRecord_VP->child));
ListFor(PersonnelRecord_VP->child)
    Encode_ChildInformation(UNIVERSAL, CONSTRUCTOR, 17,
        (PersonnelRecord_VP->child)->next->item);
Encode_struct_end(DEFINITE);

return (p);
}

File PERSONNELRECORD.decode.c:

#include "PERSONNELRECORD.defs.c"

Date  *
Decode_Date(class, form, code, B, Date_VP)
    short    class;
    short    form;
    long     code;
    IOP      *B;
    OCTS     *Date_VP;
{
    Date_VP = (OCTS *) Decode_octs(class, form, code, B, Date_VP);
    return (Date_VP);
}

EmployeeNumber  *
Decode_EmployeeNumber(class, form, code, B, EmployeeNumber_VP)
    short    class;
    short    form;
    long     code;
    IOP      *B;
    int      *EmployeeNumber_VP;
{
    EmployeeNumber_VP = (int *) Decode_int(class, form, code, B,
        EmployeeNumber_VP);
APPENDIX E. EXAMPLE OF USING CASNI AND ED LIBRARY

return (EmployeeNumber_VP);
}

Name *
Decode_Name(class, form, code, B, Name_VP)
short class;
short form;
long code;
IOP *B;
Name *Name_VP;
{
if (Name_VP == NULL)
    Name_VP = (Name *) GETBUF(sizeof(Name));

Decode_struct_beg(class, form, code, B);

Decode_octs(UNIVERSAL, PRIMITIVE, CODE_VSB_STRING, B,
    &(Name_VP->givenName));
Decode_octs(UNIVERSAL, PRIMITIVE, CODE_VSB_STRING, B,
    &(Name_VP->initial));
Decode_octs(UNIVERSAL, PRIMITIVE, CODE_VSB_STRING, B,
    &(Name_VP->familyName));

Decode_struct_end(B);

return (Name_VP);
}

Childlnformation *
Decode_ChildInformation(class, form, code, B, Childlnformation_VP)
short class;
short form;
long code;
IOP *B;
Childlnformation *Childlnformation_VP;
{
if (Childlnformation_VP == NULL)
    Childlnformation_VP = (Childlnformation *) GETBUF
        (sizeof(Childlnformation));

Decode_struct_beg(class, form, code, B);
APPENDIX E.  EXAMPLE OF USING CASN1 AND ED LIBRARY

Decode_Name(APPLICATION, CONSTRUCTOR, 1, B,
& (ChildInformation_VP->rll));
Decode_struct_beg(CONTEXT, CONSTRUCTOR, 0, B);
Decode_Date(APPLICATION, PRIMITIVE, 3, B,
& (ChildInformation_VP->dateOfBirth));
Decode_struct_end(B);
Decode_struct_end(B);
return (ChildInformation_VP);
}

PersonnelRecord *
Decode_PersonnelRecord(class, form, code, B, PersonnelRecord_VP)
short  class;
short   form;
long    code;
IOP     *B;
PersonnelRecord *PersonnelRecord_VP;
{
if (PersonnelRecord_VP == NULL)
    PersonnelRecord_VP = (PersonnelRecord *) GETBUF
    (sizeof(PersonnelRecord));
Decode_struct_beg(class, form, code, B);
Decode_Name(APPLICATION, CONSTRUCTOR, 1, B,
& (PersonnelRecord_VP->rll));
Decode_struct_beg(CONTEXT, CONSTRUCTOR, 0, B);
Decode_octs(UNIVERSAL, PRIMITIVE, CODE_VSB_STRING, B,
& (PersonnelRecord_VP->title));
Decode_struct_end(B);
Decode_EmployeeNumber(APPLICATION, PRIMITIVE, 2, B,
& (PersonnelRecord_VP->number));
Decode_struct_beg(CONTEXT, CONSTRUCTOR, 1, B);
Decode_Date(APPLICATION, PRIMITIVE, 3, B,
& (PersonnelRecord_VP->dateOfHire));
APPENDIX E. EXAMPLE OF USING CASN1 AND ED LIBRARY

Decode_struct_end(B);

Decode_struct_beg(CONTEXT, CONSTRUCTOR, 2, B);
Decode_Name(APPLICATION, CONSTRUCTOR, 1, B,
            &(PersonnelRecord_VP->nameOfSpouse));
Decode_struct_end(B);

Decode_struct_beg(CONTEXT, CONSTRUCTOR, 3, B,
            &(PersonnelRecord_VP->child));
while (ListEnd(B))
    ListAppend(&(PersonnelRecord_VP->child),
                Decode_ChildInformation(UNIVERSAL, CONSTRUCTOR, 17, B, 0));
Decode_struct_end(B);

Decode_struct_end(B);

return (PersonnelRecord_VP);
}

3. Application Program

#include "PERSONNELRECORD.defs.c"

/* ---------------------------------------------------------------
 * main - example program for PersonnelRecord encode/decode
 * ---------------------------------------------------------------
 */
main()
{
    byte      *octseq;
    IOP        bp;
    IDX        *ptr;
    int        i, len;
    PersonnelRecord value, *v;
    ChildInformation *p;

    /* ---------------------------------------------------------------
     * assign the data structure example values
     * ---------------------------------------------------------------
     */
    if (INIT != TRUE)
INITSYS();

value.rll.givenName.next = NULL;
value.rll.givenName.len = 4;
value.rll.givenName.data = (byte *) "John"; /* octet str */
value.rll.initial.next = NULL;
value.rll.initial.len = 1;
value.rll.initial.data = (byte *) "P"; /* octet str */
value.rll.familyName.next = NULL;
value.rll.familyName.len = 5;
value.rll.familyName.data = (byte *) "Smith"; /* octet str */

value.title.next = NULL;
value.title.len = 8;
value.title.data = (byte *) "Director"; /* octet str */

value.number = 51; /* INTEGER */

value.dateOfHire.next = NULL;
value.dateOfHire.len = 8;
value.dateOfHire.data = (byte *) "19710917"; /* octet str */

value.nameOfSpouse.givenName.next = NULL;
value.nameOfSpouse.givenName.len = 4;
value.nameOfSpouse.givenName.data = (byte *) "Mary"; /* octet str */

value.nameOfSpouse.initial.next = NULL;
value.nameOfSpouse.initial.len = 1;
value.nameOfSpouse.initial.data = (byte *) "T"; /* octet str */
value.nameOfSpouse.familyName.next = NULL;
value.nameOfSpouse.familyName.len = 5;
value.nameOfSpouse.familyName.data = (byte *) "Smith"; /* octet str */

p = (ChildInformation*) GETBUF(sizeof(ChildInformation));
p->rll.givenName.next = NULL;
p->rll.givenName.len = 5;
p->rll.givenName.data = (byte *) "Ralph"; /* octet string */
p->rll.initial.next = NULL;
p->rll.initial.len = 1;
p->rll.initial.data = (byte *) "T"; /* octet str */
APPENDIX E. EXAMPLE OF USING CASN1 AND ED LIBRARY

```c
p->rll.familyName.next = NULL;
p->rll.familyName.len = 5;
p->rll.familyName.data = (byte *) "Smith"; /* octet str */
p->dateOfBirth.next = NULL;
p->dateOfBirth.len = 8;
p->dateOfBirth.data = (byte *) "19571111"; /* octet str */
ListAppend(&(value.child), p);

p = (ChildInformation*) GETBUF(sizeof(ChildInformation));
p->rll.givenName.next = NULL;
p->rll.givenName.len = 5;
p->rll.givenName.data = (byte *) "Susan"; /* octet string */
p->rll.initial.next = NULL;
p->rll.initial.len = 1;
p->rll.initial.data= (byte *) "B"; /* octet str */
p->rll.familyName.next = NULL;
p->rll.familyName.len = 5;
p->rll.familyName.data = (byte *) "Jones"; /* octet str */
p->dateOfBirth.next = NULL;
p->dateOfBirth.len = 8;
p->dateOfBirth.data = (byte *) "19590717"; /* octet str */
ListAppend(&(value.child), p);

printf("Original value of PersonnelRecord:\n\n");
printREC(&(value));

/*
 * --------------------------------------------------------
 * Encode the data struct by ED encode routines
 * --------------------------------------------------------
 */

ptr = (IDX *) Encode_PersonnelRecord
    (APPLICATION, CONSTRUCTOR, 0, &(value));

/*
 * --------------------------------------------------------
 * Serilize the IDX link into octet sequence and print it.
 * --------------------------------------------------------
 */
APPENDIX E. EXAMPLE OF USING CASN1 AND ED LIBRARY

octseq = (byte *) serIDX(ptr, &octseq, &len);
printf("\n\nTransfer Syntax of PersonnelRecord:\n");
prints(octseq, len, TRUE);

/*
* Decode the octet sequence and print the produced data
*------------------------------------------------------
*/
bp.b = octseq;
bp.f = NULL;
bp.fname = NULL;

v = (PersonnelRecord *) Decode_PersonnelRecord
   (APPLICATION, CONSTRUCTOR, 0, &bp, 0);
printf("\nDecoded value of PersonnelRecord:\n");
printREC(v);

/*
* release allocated memory space
*------------------------------------------------------
*/
RESETMEMO();
}

/* printREC - print data structure PersonnelRecord
*------------------------------------------------------
*/
printREC(m) /* print out the PersonnelRecord */
PersonnelRecord *m;
{
    int i;

    printf("PersonnelRecrd{\n");
    printf("    Name {\n");
printf("  givenName  : ");
pocts(&(m->rll.givenName));
printf("  initial  : ");
pocts(&(m->rll.initial));
printf("  familyName  : ");
pocts(&(m->rll.familyName));
printf("  >
" );
printf("  title  : ");
pocts(&(m->title));

printf("  EmployeeNumber : %d\n", m->number);

printf("  dateOfHire  : ");
pocts(&(m->dateOfHire));

printf("  nameOfSpouse{\n" );
printf("  givenName : ");
pocts(&(m->nameOfSpouse.givenName));
printf("  initial : ");
pocts(&(m->nameOfSpouse.initial));
printf("  familyName : ");
pocts(&(m->nameOfSpouse.familyName));
printf("  }\n" );

printf("  ChildInformation{\n" );
ListFirst((m->child));
ListFor(m->child){
printf("  {\n" );
printf("  Name {\n" );
printf("  givenName : ");
pocts(&(m->child->next->item->rll.givenName));
printf("  initial : ");
pocts(&(m->child->next->item->rll.initial));
printf("  familyName: ");
pocts(&(m->child->next->item->rll.familyName));
printf("  }\n" );
printf("  dateOfBirth : " );
pocts(&(m->child->next->item->dateOfBirth));
printf("  }\n  }\n" );
}
4. Application Program Compilation Makefile

```makefile
OBJS = PERSONNELRECORD.encode.o PERSONNELRECORD.decode.o \\
/spring/yang/the/ED/o/*o example.o

example : $(OBJS)
    cc -g -o example $(OBJS)

example.o : example.c
    cc -g -c example.c

PERSONNELRECORD.encode.o: PERSONNELRECORD.encode.c
    cc -g -c PERSONNELRECORD.encode.c

PERSONNELRECORD.decode.o: PERSONNELRECORD.decode.c
    cc -g -c PERSONNELRECORD.decode.c
```

5. Application Program Execution Script

Original value of PersonnelRecord:

PersonnelRecord{
    Name {
        givenName : John (4)
        initial   : P (1)
        familyName : Smith (5)
    }
    title   : Director (8)
    EmployeeNumber : 51
    dateOfHire : 19710917 (8)
    nameOfSpouse{
        givenName : Mary (4)
        initial   : T (1)
        familyName : Smith (5)
    }
    ChildInformation{
        Name {
            givenName : Ralph (5)
            initial   : T (1)
            familyName: Smith (5)
        }
    }
}
APPENDIX E. EXAMPLE OF USING CASN1 AND ED LIBRARY

```
dateOfBirth : 19571111 (8)
{
    Name {
        givenName : Susan (5)
        initial : B (1)
        familyName : Jones (5)
    }
    dateOfBirth : 19590717 (8)
}
```

Transfer Syntax of PersonnelRecord:

```
60 81 85 61 10 1a 4 4a 6f 68 6e 1a 1 50 1a 5 53 6d 69 74 68
    a John P Smith
a0 a 1a 8 44 69 72 65 63 74 6f 72 42 1 33 a1 a 43 8 31 39
    Director B 3 C 1 9
37 31 30 39 31 37 a2 12 61 10 1a 4 4d 61 72 79 1a 1 54 1a 5
    7 1 0 9 1 7 a Mary T
53 6d 69 74 68 a3 42 31 1f 61 11 1a 5 52 61 6c 70 68 1a 1 54
    Smith B 1 a Ralph T
1a 5 53 6d 69 74 68 a0 a 43 8 31 39 35 37 31 31 31 31 31 1f
    Smith C 1 9 5 7 1 1 1 1 1
61 11 1a 5 53 75 73 61 6e 1a 1 42 1a 5 4a 6f 6e 65 73 a0 a
    a Susan B Jones
43 8 31 39 35 39 30 37 31 37
C 1 9 5 9 0 7 1 7
```

Decoded value of PersonnelRecord:
PersonnelRecrd{
Name {
    givenName : John (4)
    initial : P (1)
    familyName : Smith (5)
}
title : Director (8)
EmployeeNumber : 51
dateOfHire : 19710917 (8)
nameOfSpouse{
    givenName : Mary (4)
    initial : T (1)
    familyName : Smith (5)
}
ChildInformation{
    {
        Name {
            givenName : Ralph (5)
            initial : T (1)
            familyName : Smith (5)
        }
        dateOfBirth : 19571111 (8)
    }
    {
        Name {
            givenName : Susan (5)
            initial : B (1)
            familyName : Jones (5)
        }
        dateOfBirth : 19590717 (8)
    }
}