HOW FAST CAN ASN.1 ENCODING RULES GO?

By

Michael Sample

B. Sc. (Computer Science) University of British Columbia

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

COMPUTER SCIENCE

We accept this thesis as conforming

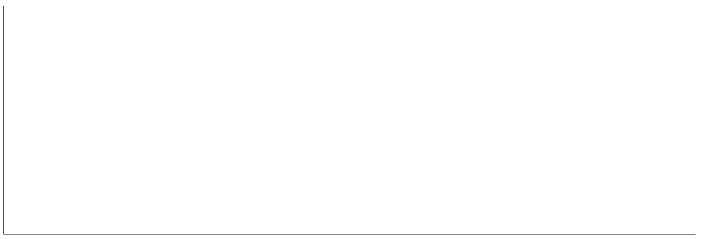
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April 1993

© Michael Sample, 1993

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.



Computer Science
The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1Z1

Date: April 30, 1993

Abstract

The tasks of encoding and decoding complex data structures for network transmission are more expensive in terms of processor time and memory usage than any other components of the protocol stack. This problem can be partially addressed by simplifying the network data encoding rules and streamlining their implementation.

We examined the performance of four network data representation standards: ASN.1 Basic Encoding Rules (BER) and Packed Encoding Rules (PER), Sun Microsystem's External Data Representation (XDR) and HP/Apollo's Network Data Representation (NDR). The sizes of the encoded values produced by each set of encoding rules are compared and several implementations are measured for code size and throughput.

To help examine implementation issues, we designed the *snacc* ASN.1 compiler that produces compile-based (C and C++) and table-based encoders and decoders as well as type tables. The performance of the *snacc*, UBC CASN1(BER), NIDL(NDR), rpcgen(XDR) and ISODE(BER) tools was examined as well as a hand-coded PER implementation. The implementation issues investigated include compiled versus table-based encoding and decoding, implementation language (C/C++), buffering techniques and memory management.

We found that the areas crucial to efficient encoder and decoder implementations are memory management, buffer management and the overall simplicity of the encoding rules. Table-based encoders and decoders typically have considerably smaller code size but perform less efficiently than their compiled counterparts. In contrast to popular belief, it is possible to implement ASN.1 BER and PER encoders and decoders that perform as well as their NDR and XDR counterparts.

Table of Contents

List of Tables

List of Figures

Acknowledgement

I would like to thank my loving parents for support and encouragement throughout my scholastic endeavors. In particular, tolerance of my running off to play in the mountains when I should have been studying.

I would also like to thank Gerald Neufeld, my supervisor and friend, for his guidance and ideas. Without his encouragement I would not have entered the M. Sc. program. Thanks also to Norm Hutchinson for plowing through this thesis.

Many thanks to Barry Brachman for the suggestions, X.500 data, incredible proof reading and the jam sessions. Don Acton provided huge amounts of help with obscure features of Latex, Emacs, Foiltex and other software packages. I also appreciated the help and support of other friends in the department: Carlin, Frank, John, George, Ian, Mike, Ming, Moyra, Murray, Peter, Stuart and Terry.

Thanks to the office staff: Carol, Evelyn, Gale, Grace, Joyce, Katie, Monica and Sunnie for cutting through the UBC bureaucracy and handling last minute courier packages.

For Rick

Chapter 1

Introduction

Computer networks spawned the need for a mechanism to translate data between machine architectures and applications. This translation process can be a significant part of the time it takes to process received protocol data units (PDUs) [Clark89]. In addition, the amazing speeds at which newer networks based on asynchronous transfer mode (ATM) or fiber distributed data interchange (FDDI) can deliver data to computers has increased the need for high performance translation mechanisms.

Using faster processors will help, but the gap between processor speed and network speed is large. For example, if a computer receives a burst of data from a 150 Mbps ATM connection, each byte of data must be processed in only 53 nanoseconds. This is less than the memory cycle time on most workstations. To make matters worse, typical protocol implementations make one or more copies of the PDU's data as it moves up the protocol stack.

Flow control is another possible technique to avoid overwhelming a receiving host. Flow control is a difficult issue because traffic can vary so much that a pessimistic flow control mechanism may limit the network usage to much below its maximum bandwidth. Optimistic flow control that allows bursts of traffic will not solve the problem.

There is no obvious single solution to the protocol performance problem. However, we can start by improving the performance of time-consuming components of computer communication, such as data translation to minimize the chances of overwhelming a networked computer with too much data. This thesis examines the optimization of the

data translation part of computer communications.

1.1 The Need for Network Data Representations

The need for network data representations has been around since computers were connected to networks. A canonical representation enables computers with different architectures to exchange data and typically simplifies data transmission by grouping or "marshalling" the data into a single logical block. Within the OSI protocol stack, encoding from a local machine representation to a canonical representation (as well as decoding) is done at the Presentation Layer [OSI88].

The need can be illustrated with the simple example of the big versus little endian representation of integers. Consider an MC68000 based machine (big endian) and an Intel80486 based machine (little endian) on a network. By sending the integer 17 in its MC68000 representation from the MC68000 to the Intel80486 without translation, the number will be interpreted as 16842752 on the Intel80486. This is clearly undesirable.

There are many representation issues involving integers alone, such as 1's versus 2's complement or 2 versus 4 byte or larger integers. Other types, such as real numbers and character strings (e.g. ASCII or EBCDIC), add to the translation problems.

Converting simple types such as integers and strings is not enough. Data grouping concepts like linked lists, structures (`struct`) in the C programming language and Pascal records need to be supported as well.

There have been many data representation standards defined in the last two decades. These include:

- ISO/CCITT's Basic Encoding Rules (BER) [BER88]
- ISO/CCITT's Packed Encoding Rules (PER) [PER92]

- Sun's eXternal Data Representation (XDR) [Sun87]
- HP/Apollo's Network Data Representation (NDR) [Kong90]
- Xerox's Courier [Xerox81]
- NIST's Computer Based Message System [NBS83]
- Multi-Media Mail [Postel80]

These encoding rules all solve the same problems. The types they support are much the same except that Multi-Media Mail proposed a shared reference type that is missing from the others. A shared reference type is potentially useful for complex data structures that have indexes and cycles.

1.2 The Performance Problem

The advent of high speed networks such as FDDI and ATM has moved the performance bottleneck into the higher layers of the protocol stack. Work on optimizing transport protocols indicates that some of the most expensive parts of a protocol implementation are buffer management and memory copying [Clark89]. These problems also exist in the Presentation Layer.

Perhaps the most expensive part of the OSI protocol stack is the Basic Encoding Rules (BER) encoding and decoding done in the presentation layer. A substantial amount of the protocol stack's processing total time can be used for encoding and decoding alone [Clark90]. There are several possible ways to reduce this overhead:

- use non-standard data representations tuned for the application
- use non-standard data representations tuned for the implementation

- design and use lightweight encoding rules
- optimize implementations of existing encoding rules

Data representations that are tuned for a particular application or protocol can be very efficient. TCP and IP packet definitions are examples of this technique. Unfortunately, this method becomes cumbersome with more complex data structures that have many optional or variable-sized components.

Assumptions based on the implementation and target environment can be used to optimize encoding rules. For example, if the target environment is a network of Sun workstations and the application is written in the same language on each host, complex data structures may be exchanged simply by grouping the components and translating pointers to offsets. This method works efficiently but is fragile with respect to the architecture of the communicating hosts; it is not a general purpose solution. Adding a machine with a different architecture to the network, such as a Intel80486 machine, would cause problems. This solution also assumes that the data structure used by the sender and receiver are identical.

Using lightweight encoding rules is a reasonable approach [Huitema89], however, some flexibility may be sacrificed (e.g. using fixed-size integers) to simplify them. Optimizing implementations of the existing general purpose encoding rules, if possible, is a desirable solution as the protocol and application designers benefit from the rules' flexibility.

We did not attempt to design new encoding rules or examine the less general purpose optimization techniques. Instead, we focussed on optimizing the implementation of general purpose encoding rules such as BER.

1.3 Our Research Goal

We examined the performance of four network data representation standards: BER, PER, XDR and NDR. BER, XDR and NDR are popular encoding rules. PER [PER92] have recently been introduced and are likely to be widely used. We looked at the size of the encoded values and the performance of several implementations to determine the critical implementation issues.

The size of an encoded value generated by a given set of encoding rules can be important when dealing with networks that are slow or have a high cost per packet. Other considerations include storing data on machines with limited disk space. With disks and memory becoming cheaper and networks becoming faster, a more important consideration is throughput.

The throughput of several implementations of BER, XDR and NDR and a hand coded PER implementation is measured. The object code size and dynamic memory usage of these implementations is also examined. While the dollar cost of RAM memory is small these days, code size and memory usage must be considered for their impact on throughput.

To help examine implementation issues, we implemented the *snacc* (Sample Neufeld ASN.1 to C/C++ Compiler) ASN.1 compiler which produces C and C++ BER encoders and decoders as well as type tables. The performance of the *snacc*, UBC CASN1(BER), NIDL(NDR), rpcgen(XDR) and ISODE(BER) tools is examined, as well as a hand-coded PER implementation. The implementation issues investigated include compiled versus table-based encoding and decoding, implementation language (C/C++), buffering techniques and memory management.

This thesis assumes the reader is familiar with ASN.1 [ASN188], BER [BER88], XDR [Sun87] and NDR [Kong90]. Several ASN.1 and BER tutorials are available [Steedman90]

[Neufeld92-1].

The next chapter introduces the concept of data definition languages and data formats, covering the ASN.1, XDR and NIDL languages and the BER, PER, XDR and NDR encoding rules. PER are described in depth since they are a very new standard [PER92].

In Chapter three we survey previous efforts to improve the performance of encoding rules. The performance-related aspects of the *snacc* compiler: its implementation and the code it generates, are given in Chapters four and five. The metrics used to benchmark the encoding rules, the tools that implement them and an analysis of the results are given in Chapter six. The final chapter contains the conclusions drawn from our research. The appendix contains a C and C++ example of *snacc* generated code and the type table's ASN.1 data structure definition.

For those familiar with ASN.1, XDR and NDR, Chapters six and seven will likely be the most interesting. Implementors may wish to focus on Chapters three, four and six as well as the app endices. Users interested in using the *snacc* compiler sho user manual [Sample93-1].

Chapter 2

Data Definition Languages and Encoding Rules

Data definition languages are used to specify the content and structure of data. The encoding rules specify how a particular value of a data structure will be represented. A data definition language is referred to as an abstract syntax notation. The transfer syntax defines the way a particular instance of the application data structure is encoded. A particular data structure in the abstract syntax notation is referred to as an abstract syntax.

An example will clarify these definitions. If you wanted to specify a data structure that contained the date (year, month, day), the abstract syntax in ASN.1 might look like:

```
Date ::= SEQUENCE
{
   year  INTEGER,
   month INTEGER (1..12),
   day   INTEGER (1..31)
}
```

The Date value "May 1, 1993" encoded according to the BER transfer syntax is (shown in hexadecimal):

```
300a020207c9020105020101
```

All of the transfer syntaxes (encoding rules) that we tested grouped the represented values into a single logical block. The grouped or marshalled data can easily be passed to the transport layer.

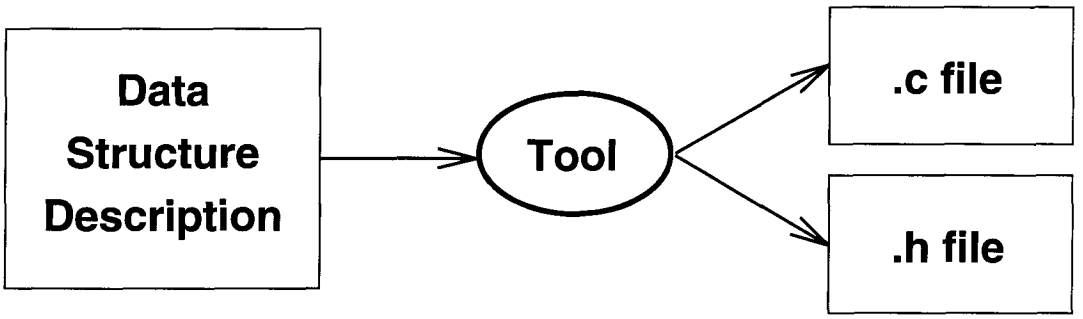


Figure 2.1: A typical compilation process

Typically the data definition languages are designed to be processed by a compiler that generates encoders, decoders and other utility routines for the given data structure in a target programming language such as C or Pascal (see Figure 2.1). Other approaches include extending a programming language's grammar to handle the data definition language internally [Bochman89].

Data languages such as ASN.1 are used to define protocol data units (PDUs) in protocol standards such as X.400 [MHS88] and X.500 [DS88]. The protocol definition documents for protocols such as TCP and IP use ASCII drawings (see Figure 2.2) and text that describe both the type content (abstract syntax) and representation (transfer syntax) of their PDUs.

Using data definition languages with encoding rules separates the abstract syntax from the transfer syntax. The data descriptions are more formal, with less chance for ambiguity. Once a data definition language has been learned, it provides a high level way to think about and design data structures. Another benefit of data languages is that they can be compiled for automatic implementation of encoders, decoders and other routines.

The advantages of the TCP/IP style of PDU definition is that the definitions are self contained (no other documents or language understanding is needed) and the PDUs can be very compact. Also, by placing the fixed-sized components first, the decoding cost is reduced or eliminated. The disadvantage is that the use of fixed-size types limits their

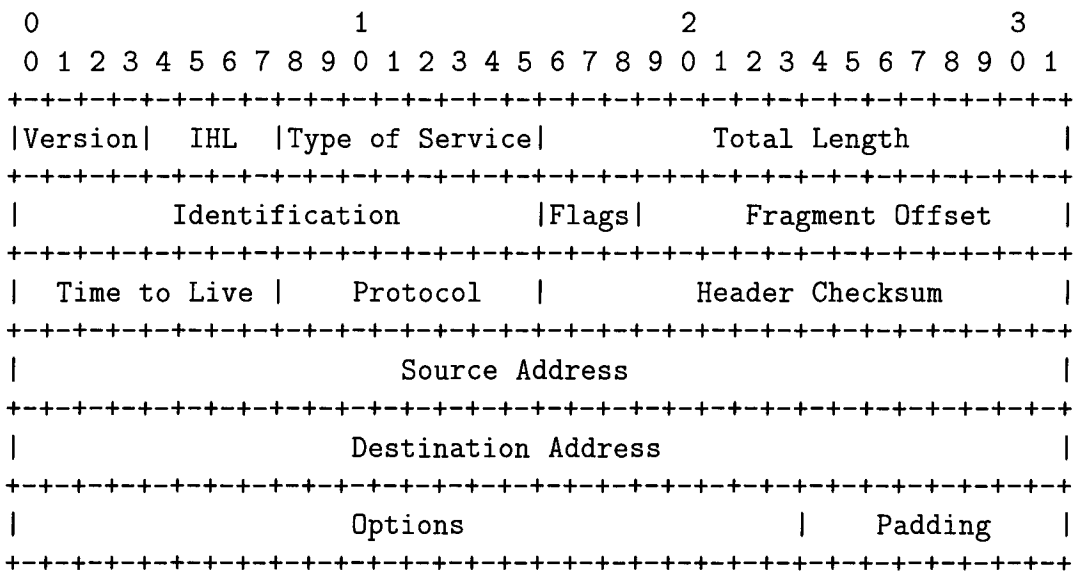


Figure 2.2: An Internet Protocol (IP) header description

expansion possibilities (e.g. the IP address space is now too small and cannot easily grow). Also, this style of PDU definition becomes very cumbersome when the number of components is large, especially if many of the components are not fixed in size.

2.1 Data Definition Languages

There are two main uses for data structures defined in a standardized data language. One purpose is to specify the content and structure of a PDU for a certain protocol. For example, the protocol definition documents for X.500, X.400, SNMP [snmp90], Kerberos [Steiner88] [Kohl92] and WAIS [Z39.50] use ASN.1 to define their PDUs.

The other use of the data specifications is to implement part of the protocol. Special-purpose compilers are usually used to automatically generate the encode, decode and utility routines for the PDUs.

The languages in our performance test used two different approaches to define the

language. NIDL and XDR are defined very closely to an existing programming language, C, to simplify their usage by programmers already familiar with C. This advantage is offset by the fact that it biases the protocol implementation towards a particular language. This problem is highlighted by the fact that NIDL has a Pascal variant in addition to the C version.

The data structures defined in these languages have extra information, such as whether a component of a `struct` is referenced by a pointer. This is a local data representation issue that should not be dictated by a protocol standard. The implementation language bias and local representation information in XDR and NIDL make them unsuitable for heterogeneous protocol specification.

ASN.1 takes a different approach. The key concepts of data structures were abstracted and separated from other implementation-specific concepts such as pointers. The fact that ASN.1 did not evolve from an implementation language such as C leaves it free to change independently of the implementation language. The disadvantage of the ASN.1 approach lies in the fact that implementors need to become familiar with a completely new and complex language.

2.2 General Aspects of Encoding Rules

There are several approaches to network data representation:

- a single standardized data format
- data format defined by sending host
- data format defined by receiving host

Using a single standardized format means that each host only needs a single set of encoding and decoding routines for each primitive type. The other techniques may be

optimal if the communicating hosts have the same architecture, but in general more encoding and decoding routines will be needed. Specifically, if there are N machine formats and the data format is defined by the sending host, each host will need $N - 1$ decoding routines.

Encoding rules can support *self-defining* values. Self-defining means that type information is included in the values. Other terms that have been used to describe this include explicit (self-defining) and implicit type representation [DeShon86] [Partridge89]. Self defining values allow the implementation of "generic" decoders, useful for implementations and protocol testing.

One-pass encoding and decoding is defined as conversion between the local (non-standardized) and transfer representation that can be done in one step, without using intermediate representations. This improves performance.

Some encoding rules can produce *canonical* encodings where each abstract value maps to exactly one encoding. This can be useful for generating digital signatures for security purposes. A simple example of this is the encoding of boolean values. Some encoding rules allow "true" to be encoded as any non-zero integer. The fact that "true" can be encoded many ways implies that the encoding rules are not canonical.

Encoding rules that have *relay safe* encodings means that encoded values can be relayed or otherwise handled by systems that do not know the values' abstract syntaxes.

It is important to distinguish between *forward* and *backward* encoding and decoding techniques. In forward encoding, the first component of the encoded value is calculated and written first with the following components appended to it. Backward encoding is the process of writing the last component of an encoded value and prepending the preceding components. The same terms can be applied to decoding, however, all of the encoding rules and implementations we tested used forward decoders.

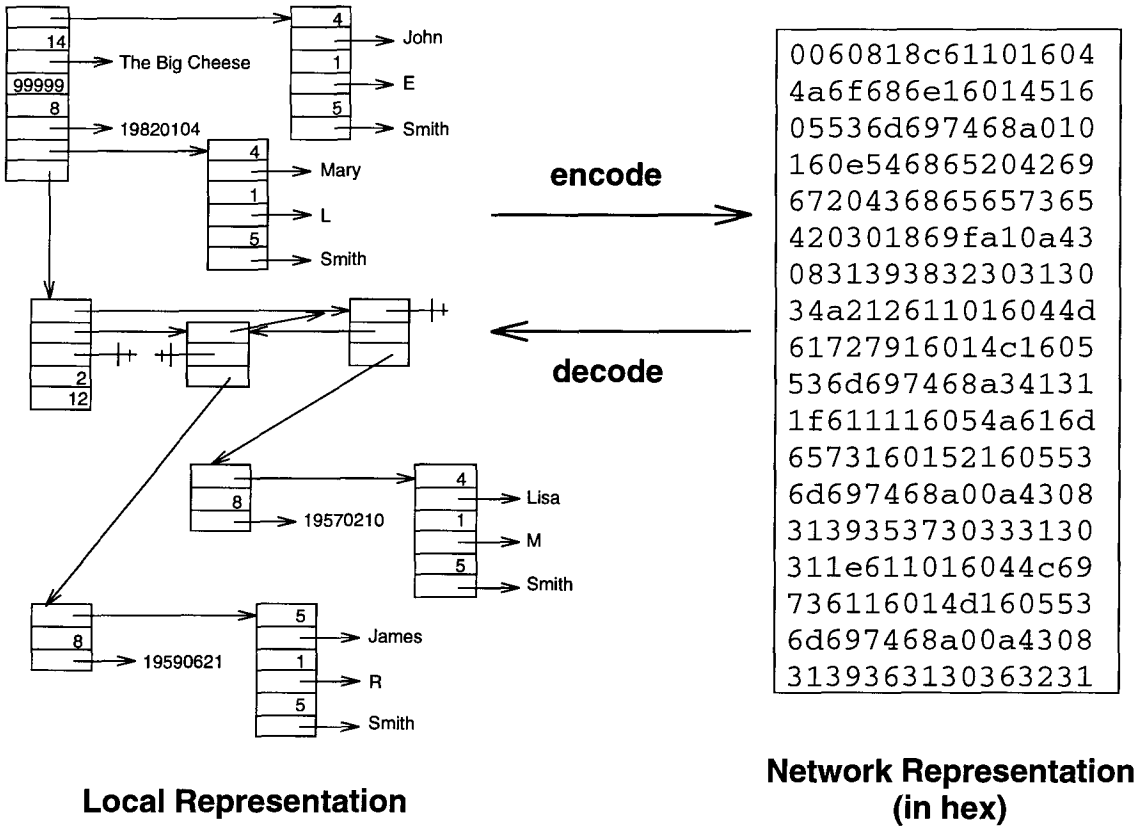


Figure 2.3: The encoding and decoding process

Primitive types are types such as integers that are not composed of other types. Constructed types are types such as lists and structures that are composed of one or more primitive or constructed types. We shall refer to the types that make up a constructed type as its components.

Another aspect of encoding rules that should be considered is their expected longevity. Will they need to be modified as architectures move from 32 to 64 bit words and beyond?

Ideal encoding rules would be simple, canonical, relay safe and support fast encoding and decoding as well as compact encodings.

2.3 General Encoding and Decoding Procedures

Encoding is the translation of a local representation of a data value to its network format (see Figure 2.3). Generally, encoding consists of three parts:

- encoding primitive values into their network format
- allocating a send buffer as necessary
- writing the data in its network format into the send buffer

Decoding is the process of converting a value from the transfer or external format into a local representation of the value. There are three main parts to the decoding process:

- reading from the receive buffer
- decoding primitive data in its network format into its local representation
- allocating space for the decoded value

The goal is to optimize each of these steps. The choice of buffer and memory management schemes is very important as the encoding and decoding processes can make heavy use of their services. The other consideration is the cost of translating the primitive types between the local and network format.

2.4 ASN.1: Abstract Syntax Notation One and its Encoding Rules

ASN.1 and BER were initially designed as part of the 1984 ISO/CCITT X.400 electronic mail standard [ASN184]. They are a flexible and robust means of defining protocol data structures and representing their values.

Since many other applications and protocols could benefit from the general purpose, robust nature of ASN.1/BER, they were moved out of the X.400 standard and became

separate standards in 1988, CCITT X.208 and X.209 (or ISO/IEC 8824 and 8825) respectively. ASN.1 and BER were split into two separate standards for the reason that more that one set of encoding rules could be used with ASN.1.

Between 1984 and 1988 many features were added to the ASN.1 language, such as subtyping, a powerful type constraining mechanism. The 1992 version of ASN.1 solves some of the parsing and semantic problems of ASN.1 '88.

2.4.1 ASN.1: The language

ASN.1 is a rich language that defines types. It does not contain imperative semantics of typical programming languages nor is its syntax similar to any popular programming language. It does allow very precise definition of types.

ASN.1 has eight basic primitive types and five constructor types. The primitive types are:

- BOOLEAN
- INTEGER
- ENUMERATED
- NULL
- REAL
- OBJECT IDENTIFIER
- OCTET STRING
- BIT STRING

The constructor types are:

- SEQUENCE
- SET
- SEQUENCE OF
- SET OF
- CHOICE

Unlike other languages, ASN.1 has more than one set of encoding rules defined for it. This allows implementors to choose encoding rules that support their needs without rewriting a protocol's ASN.1 definition.

2.4.2 BER: Basic Encoding Rules

BER produce regular tag-length-content encodings, where the content can be composed of more tag-length-content tuples or a primitive type's (e.g. INTEGER) value. The components of a BER encoding are octet (8 bit) aligned.

The tag can provide information concerning the type of the content. BER produce self-defining or explicitly typed values. The fact that type information is present in BER encodings makes BER different from all of the other encoding rules we tested.

It should be noted that if implicit tagging is used, the type information in a BER encoded value is not complete. For example, if UNIVERSAL tags are not used, a tag may tell the decoder that the value is primitive (i.e. the content does not contain more tag-length-content tuples) but not whether it is an INTEGER, BOOLEAN, REAL or other primitive type.

The regular nature of BER encodings allows the implementation of encoders and decoders without knowledge of the abstract syntax. This feature is used by some implementations to provide a common value form to pass to and receive from the presentation

layer. Some consider type information in values an unnecessary expense in time and size since protocols typically know what type of data they will receive.

The encoding techniques for the primitive values are very flexible. BER allows arbitrarily large integer and string values. In practice these sizes are limited by the architecture of the encoding host. This flexibility may increase the useful lifetime of BER.

There are two subsets of BER that provide canonical encodings. One is Distinguished Encoding Rules (DER) and the other is Canonical Encoding Rules (CER).

BER has a feature that makes it difficult to implement one-pass BER encoders. Encoding a definite length before a constructed value is difficult because the length itself is a variable in size but it is only known after the components of the constructed value have been encoded. Thus one needs a scheme to insert the length after the components have been encoded. Unfortunately this usually results in a two pass encoder. Fortunately, the indefinite length option does not have this problem. However, DER require definite lengths so other solutions such as backwards encoding need to be employed to avoid costly two pass encoding.

2.4.3 PER: Packed Encoding Rules

In this section most of the concepts of Packed Encoding Rules (PER) are described; for the complete definition see [PER92]. There are some problems with the current Committee Draft for PER that will be rectified before PER goes on to become a Draft International Standard.

PER have evolved immensely since their initial draft [PER91]. Initially, PER were basically an optimized BER that removed redundant tags and lengths as well as compressing integers and booleans under special circumstances. The current version of PER is described here.

PER were designed to produce compact encodings. Compact encodings are useful

when storing large amounts of data on disk or transmitting PDUs over a network in which the cost per byte or packet is high. The amount of compression PER achieves over BER can be very dependent on the encoded values' ASN.1 type definition.

Some aspects of ASN.1 type definitions that do not affect BER encodings will affect PER encodings. PER use subtyping value range and size restrictions to improve compression. Careful use of subtyping by ASN.1 writers can significantly reduce the size of PER encodings.

PER Differences From BER

Many people are familiar with BER; here are the fundamental differences between PER and BER.

In PER:

- Tags are not encoded.
- Subtyping information can affect the encoded value.
- The value range of ENUMERATED types can affect the encoded value.
- The values range of named bits in BIT STRINGs can affect the encoded value.
- Lengths are only encoded before non-fixed-size primitive values.
- The number of components in SET OF and SEQUENCE OF values is encoded before the content instead of the content's encoded length.
- The components of SET and SET OF types are ordered.
- Type extension must be explicitly allowed by using the "..." extensions marker in the ASN.1 definitions of the extensible types (ASN.1 '92 feature)

PER uses a preamble-length-content format, each of which may be empty. The preamble is only present for SEQUENCEs, SETs and CHOICEs to indicate which components are present. Subtyping and other information that restricts the encoded length of a value will be used to compress or eliminate the length determinant. Subtyping and other information that restricts the value range of a type will be used to compress or eliminate the encoded values of that type. Thus, the PER preamble-length-content format is context sensitive (to the ASN.1 definition), unlike the regular structure of the BER tag-length-content format. Hence generic (abstract syntax ignorant) decoders cannot be written for PER values.

Extensions Marker

ASN.1 '92 introduces the concept of the extension marker. It is briefly described here because it affects how values are encoded in PER.

The extensions marker is expected to be used by ASN.1 designers in types that will likely have components added or removed in future versions of their ASN.1 specification. The extensions marker can be used in the ASN.1 definition of SETs, SEQUENCEs, CHOICEs, BIT STRINGs with named bits and ENUMERATED types.

BER supports type extension by the use of new tags on added SET, SEQUENCE or CHOICE components and the fact that named bits and ENUMERATED values do not affect their encoding technique. This form of extension does not work with PER because tags are not encoded and the named bit and ENUMERATED value information is used to help compact the encoding (i.e. it affects the way values are encoded).

```
Colour1 ::= ENUMERATED { red(0), orange(1), yellow(2), green(3)}
Colour2 ::= ENUMERATED { red(0), orange(1), yellow(2), green(3), ...}
```

The `Colour2` type above is an extensible version of `Colour1` since it uses the "..."

extensions marker. A future version of `Colour2` could be:

```
Colour2 ::= ENUMERATED { red(0), orange(1), yellow(2), green(3), ..., blue(4)}
```

PER decoders that were designed to handle the original version of `Colour2` will be able to detect the presence of an extended `Colour2` value and deal with it accordingly. This could include discarding the PDU or something else depending on the protocol.

PER decoders for the above version of `Colour1` would not be able to decode values of the following extended version of `Colour1` because the extensions marker was not used in the original and new definition.

```
Colour1 ::= ENUMERATED { red(0), orange(1), yellow(2), green(3), blue(4)}
```

The addition of an extensions marker affects the way types are encoded (i.e. the bits on the line will be different); a decoder for `Colour2` will not be able to handle an encoding of `Colour1` and vise-versa. This is why it is recommended that types that are likely to be extended in future versions of the protocol be given an extensions marker.

Flavors of PER

There are four different flavors of PER resulting from combinations of two features: BASIC or RELAY-SAFE-CANONICAL, and ALIGNED or UNALIGNED. Each flavor has its own OBJECT IDENTIFIER for use when negotiating a presentation connection and in other situations. The OBJECT IDENTIFIERs are:

```
{ joint-iso-ccitt asn1(1) packed-encoding(3) basic(0) aligned(0) }
{ joint-iso-ccitt asn1(1) packed-encoding(3) basic(0) unaligned(0) }
{ joint-iso-ccitt asn1(1) packed-encoding(3) relay-safe-canonical(0)
aligned(0) }
{ joint-iso-ccitt asn1(1) packed-encoding(3) relay-safe-canonical(0)
unaligned(0) }
```

BASIC PER is the most general form of PER. RELAY-SAFE-CANONICAL PER is a restriction of BASIC PER that makes an encoded value both relay safe and canonical.

The UNALIGNED version of PER allows the components of the encoded value to be written without octet alignment so that no bits are wasted between the components, yielding a very compact value.

Each type is designated an "octet aligned bit field" or a "bit field". The only time alignment is performed is when appending a type that is an octet aligned bit field while using an ALIGNED flavor of PER. When alignment occurs, the existing encoding is padded with zero bits until it is an integral number of octets in length, then the type that required the alignment is appended.

The concepts of constrained whole numbers, semi-constrained whole numbers and unconstrained whole numbers are used to describe encoded lengths, INTEGERs, ENUMERATED values and other features of PER. For simplicity, they are defined here. For positive binary integers and two's complement binary integers the leading bit is the most significant and the trailing bit is the least significant.

A constrained whole number is a whole number that has both an upper and a lower bound. The upper and lower bounds can be used to reduce the number of bits needed to represent the value.

Given a constrained whole number, x, with an upper bound ub, and a lower bound lb such that $lb \leq x \leq ub$, its value range is $ub - lb + 1$. The value x will be encoded as a positive binary integer of value $x - lb$. The number of bits or octets that are used to hold x's encoded value are shown in Table 2.1.

For example, a value of type INTEGER (4..7) needs only two bits to represent its range of values. From Table 2.1 we can see that this value would be encoded as a positive binary integer in a bit field 2 bits long.

value range	**size**
1	0 bits
2 to 255	bit field of the minimum number of bits large enough to hold the value range
256	octet aligned bit field 1 octet long
257 to 65,536	octet aligned bit field 2 octets long
larger than 65,536	octet aligned field the minimum number of octets needed to hold the value

Table 2.1: PER field types and sizes for constrained whole numbers

A semi-constrained whole number is a whole number that has a lower bound but no upper bound. A semi-constrained value, x, with a lower bound lb, is encoded as a positive binary integer with value $x - lb$ in an octet aligned bit field of the minimum number of octets needed for the value.

An unconstrained whole number has no upper or lower bound. They are encoded as two's complement binary integers in an octet aligned bit field of the minimum number of octets in length for the value.

Tags

PER does not encode the tag information from ASN.1 definitions. Different mechanisms are used in the areas where BER needs tags to identify the presence of optional SET or SEQUENCE components or which CHOICE component is present. These mechanisms involve the preamble and will be discussed with their respective types.

Lengths

BER-like length determinants are only encoded before primitive values such as INTEGERs and REALs that are not a fixed-size. If the size of the encoded value is constrained

by any subtyping or other information, this information will be used to shrink the length. PER lengths may be an octet count, bit count or component count depending on the type they precede.

Lengths are either constrained or semi-constrained whole numbers. They cannot be unconstrained because lengths are always positive.

If a length is constrained and its upper bound is less then 65,536, it is encoded as a constrained whole number. For example, the length of values of type OCTET STRING (SIZE (3..6)) have a lower bound of three and an upper bound of six, which means their length will be encoded in two bits, with '00' mapping to three and so on. For all other lengths, the following rules apply. Note that the lower bound does not affect how the following lengths are encoded.

If $length < 128$, the length is encoded as the positive binary integer representing *length* in a one octet long octet aligned bit field.

Otherwise, if $128 \leq length < 16,384$, the length is encoded as the positive binary integer representing *length* in a two octet long octet aligned bit field and the most significant bit is set to one.

For lengths of 16,384 or larger, a flat fragmentation scheme is used such that the maximum length of any part of a primitive value is less than 65,536 octets.

Open Types

Open types are semantically similar to the ANY type in ASN.1 but are encoded a special way. The type contained in the open type is encoded normally and preceded by its length in bits. This length allows decoders that do not know the type contained in the open type to skip the open type's value and continue decoding the rest of the value.

The length encoded before the contained type's encoding is encoded as a semi-constrained whole number with a lower bound of zero.

BOOLEAN

BOOLEAN values are encoded in one bit long bit fields. The bit is set to one to represent TRUE and zero for FALSE. They are not preceded by a preamble or a length determinant.

INTEGER

INTEGERs are encoded as constrained, semi-constrained or unconstrained whole numbers. They never have a preamble but require a length determinant in the semi-constrained and unconstrained cases.

INTEGERs with an upper and a lower bound such as INTEGER (10..20) are encoded as constrained whole numbers. The encoded integer is preceded by its length determinant only if its value range is 65,536 or greater. This is because the length of encoded constrained whole numbers is variable when their value range is 65,536 and over.

INTEGERs, such as INTEGER (0..MAX), with a lower bound but no upper bound are encoded as semi-constrained whole numbers and are preceded by their length determinant.

INTEGERs with no constraints, such as INTEGER, are encoded as unconstrained whole numbers and preceded by their length determinant.

ENUMERATED

ENUMERATED values are typically encoded as constrained INTEGERs. To calculate the constraints, the defined ENUMERATED values are sorted in ascending order and each is assigned an index starting with 0 for the first value, 1 for the second and so on. The index of the value is encoded as a constrained integer whose constraints are INTEGER(0..MAX-INDEX), where MAX-INDEX is the highest index in the ENUMERATED type.

```
Foo1 ::= ENUMERATED { spring(7), summer(12), fall(16), winter(19) }
```

The value `summer` from `Foo1` would be encoded like the value 1 of the type INTEGER(0..3). The resulting encoding would be a bit field with the bits '01'. The bits are not preceded by a length determinant because the values of Foo1 are always 2 bits long.

```
Foo2 ::= ENUMERATED { spring(7), summer(12), fall(16), winter(19), ...}
```

If the extensions marker is present in an ENUMERATED type as in `Foo2`, the values cannot be completely constrained due to the possibility of future additions to the ENUMERATED values. In these cases, ENUMERATED values are encoded as semi-constrained INTEGERs with a lower bound of zero.

REAL

REAL values are encoded the same way as BER REAL values into an octet aligned bit field. They have no preamble and are preceded by a length determinant. When using a RELAY-SAFE-CANONICAL encoding flavor of PER, the CER[CDER92] REAL rules are used.

BIT STRING

BIT STRINGs are encoded with no preamble and in some cases will be preceded by a length determinant. The upper bound ub and lower bound lb on a BIT STRING's length, as determined from SIZE subtyping or named bits, affect the way they are encoded.

BIT STRINGs with named bits are special because the named bits constrain the encoded size of the BIT STRING much like a SIZE constraint. Using n to represent the highest named bit number, BIT STRINGs with named bits are constrained as though they were a BIT STRING (SIZE(0..(n+1))). For the `FooBits` type, $lb = 0$ and $ub = 20$.

type	lower bound	upper bound
BIT STRING	0	unset
BIT STRING (SIZE 6)	6	6
BIT STRING (SIZE (3..20))	3	20
BIT STRING {night(0), day(1)}	0	2
BIT STRING {night(0), day(1),...}	0	unset

Table 2.2: Examples of length upper and lower bounds for BIT STRING types

If an extensions marker is present in the named bits, then the upper bound on size is undefined ("unset") instead of $(n + 1)$. Values of BIT STRING types with named bits have their trailing zero bits removed.

```
FooBits ::= BIT STRING {spring(7), summer(12), fall(16), winter(19)}
```

The following rules are used to encode BIT STRING values. If *ub* equals zero, the encoding is empty, with no length determinant.

If *lb* equals *ub* and *lb* is less than or equal to 16 bits, the bits are encoded in a bit field of length *lb* bits without a length determinant.

If *lb* equals *ub* and *lb* is greater than 16 bits, the bits are encoded in an octet aligned bit field of length *lb* bits without a length determinant.

Otherwise, the bits are encoded in an octet aligned bit field preceded by the length determinant in bits as constrained by *lb* and *ub*. There is no unused bits octet as in BER.

OCTET STRING

OCTET STRINGs are encoded in a way similar to BIT STRINGs. They are encoded with no preamble and in some cases will be preceded by a length determinant. As with BIT STRINGS, the OCTET STRING encoding depends on the upper bound *ub*, and lower bound *lb* of its size.

type	lower bound	upper bound
OCTET STRING	0	unset
OCTET STRING (SIZE 6)	6	6
OCTET STRING (SIZE (0..20))	0	20

Table 2.3: Examples of upper and lower bounds for OCTET STRING types

If *ub* equals zero, the encoding is empty, with no length determinant.

If *lb* equals *ub* and *lb* is less than or equal to two octets, the octets are encoded in a bit field of length *lb* octets and are not preceded by a length determinant.

If *lb* equals *ub* and *lb* is greater than two octets, the octets are encoded in an octet aligned bit field of length *lb* octets and are preceded by length a determinant.

Otherwise, the octets are encoding an octet aligned bit field preceded by the length determinant as constrained by *lb* and *ub*.

OBJECT IDENTIFIER

PER OBJECT IDENTIFIERs are encoded into an octet aligned bit field in the same way as BER OBJECT IDENTIFIERs. They have no preamble and are always preceded by their length determinant.

NULL

Nothing is encoded for a NULL value (no preamble, length or content).

SEQUENCE

SEQUENCE values usually have a preamble, never have a length determinant and their content is their encoded components in the order of their declaration.

The SEQUENCE preamble contains information about extensions to the SEQUENCE type definition and the presence of OPTIONAL and DEFAULT components.

If a SEQUENCE's ASN.1 definition has an extensions marker ("..."), the first bit of the preable indicates whether components that were not in the original SEQUENCE declaration are present in the encoding. If extension components are present in the encoding, this bit is one otherwise it is zero.

The following bits in the preamble are for the optional or default elements of the SEQUENCE, if any. There is one bit for each optional/default element and they are in the order of their declaration. A value of 1 indicates that the element is present and a 0 means that the element is absent. Optional and default elements that are after the extensions marker (if any) do not have bits in the preamble. If there are more than 65,536 optional or default elements, fragmentation is used for the preamble.

The preamble is written as a bit field and followed by the encodings of each of the SEQUENCE's components in the order they were declared in.

In the RELAY-SAFE-CANONICAL case, default values will always be included in the encoding with their bit in the preamble set to one.

There is a special procedure for encoding the components declared after the extensions marker. There is a "presence" bit string that has a bit for each extension component; it is encoded as if it were a BIT STRING (SIZE (1..MAX)). The extension components are encoded as open types. As before, default values will always be included in the encoding with their bit in the preamble set to one for the RELAY-SAFE-CANONICAL cases.

SET

SETs are encoded the same way as SEQUENCEs after ordering the SET's components according to their tag values. This is different from BER, where SET components can be encoded in any order. The ordering in PER removes the need for tags.

type	lower bound	upper bound
SEQUENCE OF Foo	0	unset
SEQUENCE (SIZE 10) OF Foo	10	10
SEQUENCE (SIZE (1..10)) OF Foo	1	10

Table 2.4: Examples of upper and lower bounds for SEQUENCE OF types

First, a SET's component's tags are sorted by class with UNIVERSAL first, followed by APPLICATION, CONTEXT and PRIVATE. Within each of the class groupings, the tags are sorted in ascending order of tag code. This order defines the order of a SET's components such that the SET can be treated as a SEQUENCE.

For example, the SET `Bar1` would be encoded in the same was as the SEQUENCE `Bar2`.

```
Bar1 ::= SET
{
    a [1] INTEGER,
    b INTEGER,
    c [0] INTEGER
}

Bar2 ::= SEQUENCE
{
    b INTEGER,
    c [0] INTEGER,
    a [1] INTEGER
}
```

SEQUENCE OF

The SEQUENCE OF type's encoding consists of the number of components followed by the ordered encoding of each component. Upper and lower bounds on the number of components can affect how the number of components is encoded.

If the number of components in the SEQUENCE OF type is constant (upper bound equals the lower bound), the number of components is not encoded. Otherwise, the number of components is encoded as a length determinant. If the number of components exceeds 65,536, the SEQUENCE OF encoding will be fragmented.

SET OF

The SET OF type is treated as if it were a SEQUENCE OF type. This means that the ordering of SET OF types is significant, unlike BER.

CHOICE

In BER, tags are used to indicate which CHOICE component is present. In PER, the index of the component is used instead. First, the CHOICE components are sorted by their tags in the same manner as SET components. The first component is given the index zero, the second component's index is one and so on. CHOICEs are encoded as the index of the present CHOICE component followed by the encoded component.

If there is no extensions marker in the CHOICE's definition, the index is encoded as a constrained whole number with a lower bound of zero and upper bound equal to the index of the last component. The component is encoded as PER defines for its type and appended to the encoded index.

If the CHOICE's definition contains an extensions marker, the index is encoded as a semi-constrained whole number with a lower bound of zero. If the component from the extensions root (i.e. it is declared before the "..." symbol in the CHOICE), then it is encoded normally and appended to the encoded index. If the component is in the extension additions (i.e. declared after the "..."), it is encoded as an open type.

Implementation Comments

Examination of PER shows that one pass, forward encoding and decoding can be supported. There appear to be no problems, such as BER's definite lengths for constructed values, that require more than one pass. It is possible that the fragmentation required for PER open types whose contained encoding is longer than 65,536 bits may require more than one pass.

In most cases, the tag sorting operations for SETs and CHOICEs can be done at the time the ASN.1 is compiled. No overhead needs to be incurred during the encoding and decoding processes. Types whose first tag depends on their value (untagged CHOICE and untagged ANY types) are the exceptions that may require some runtime sorting.

The potentially unaligned nature of PER encodings requires more powerful buffer management routines to handle arbitrary bit alignment.

The lack of tags in PER simplifies code design. For BER implementations, tags are difficult to deal with since a class and code of a type's tag is set by its enclosing type (sometimes) but the tag's form is handled by the type itself (see Section 4.3). This can lead to clumsy implementations.

ASN.1 compilers will need to do more work to generate PER encoders and decoders, since more aspects of ASN.1, such as subtyping, affect PER. BER encoders and decoders can be generated without using the subtyping information.

PER Summary

PER are an alternative to BER when more compact encodings are desired, however they are less flexible. PER encodings cannot be decoded without knowing the original abstract syntax. That is, PER encodings are not self-defining. There are four types of PER, based on two factors: BASIC-PER versus RELAY-SAFE-CANONICAL PER, and ALIGNED

versus UNALIGNED. The RELAY-SAFE-CANONICAL PER is a restriction on some options in BASIC PER such that a given value has exactly one encoding. This is a useful feature for generating digital signatures. The UNALIGNED version of PER allows the components of the encoded value to be written without octet alignment so that no bits are wasted between the components, yielding a very compact transfer sytnax.

SETs are encoded the same way as SEQUENCEs after their elements have been sorted according to their tag's class and code; note that this is a compile time operation (in most cases) and therefore does not decrease performance. SET OF values are encoded as if they were SEQUENCE OF values.

Tags are not encoded at all. Bitmaps before SEQUENCEs and SETs indicate the presence of OPTIONAL and DEFAULT elements. For CHOICEs, the elements are sorted by tags like SETs and each element is then assigned an index starting from zero. This index is encoded before the CHOICE to indicate which element is present.

Only some primitive values will have their length encoded. Constructed values do not have their length encoded; instead they rely on their components to define their length. SET OF and SEQUENCE OF values have the number of components they contain encoded before the components. For large OCTET STRING and BIT STRING values, a general purpose, flat fragmentation scheme is used that allows the fragments to be a maximum of 64K bytes long.

Subtyping information is used extensively by PER to reduce the size of the encoded value. Specifically, value range information on INTEGERs and ENUMERATED types, and size information on other types is used. For example, if an OCTET STRING is a fixed-size, its length is not encoded. For ENUMERATED types and INTEGERS with value ranges, the minimum number of bits that can represent the required range of values is usually used.

2.5 XDR: External Data Representation

2.5.1 The XDR language

XDR [Sun87] was developed around 1984 to help implement Sun's Network File System (*NFS*).

Rpcgen is Sun Microsystems' RPC (Remote Procedure Call) stub generator. It parses remote procedure declarations and C-like data structures written in the XDR language and produces the C data structures, RPC stubs and the XDR encoder and decoder source code for the given data structures.

The rpcgen or XDR language is similar to C data structure definitions. This simplifies implementing a protocol in C.

2.5.2 XDR Encoding Rules

XDR provides a set of encoding rules that operate optimally if the encoding and decoding hosts have 4 byte words, big endian integers and use the IEEE floating point representation.

The eleven XDR primitive types are:

- int - 4 byte long integer, big endian 2's complement
- unsigned int - 4 byte long integer, big endian, positive binary number
- hyper - 8 byte long integer, big endian, 2's complement
- unsigned hyper - 8 byte long integer, big endian, positive binary number
- enum - encoded as an int
- bool - encoded as an int, 0 for false, 1 for true

- float - 4 bytes, IEEE float
- double - 8 bytes, IEEE double
- opaque - fixed or variable length uninterpreted data
- string - fixed or variable length ASCII data
- void - not encoded

The three XDR constructor types are:

- array - fixed or variable length array of variable or fixed-size elements
- struct
- discriminated union

XDR uses a "Basic Block Size". This means each component of an XDR encoding is aligned to a 4 byte boundary to simplify decoding on machines with this architecture. The alignment requirement may introduce padding bytes. Any padding bytes that are necessary to maintain the basic block size are set to zeros. This and the careful design of other parts of XDR, such as the `bool` boolean type, make XDR encodings canonical.

The typing information in XDR is implicit. If a protocol needs explicit type information, one can easily modify an XDR specification to contain it.

2.6 NIDL/NDR

2.6.1 NIDL: Network Interface Definition Language

Apollo Computer's NIDL compiler for C is similar to rpcgen except that NDR encoders are generated. There is also a different NIDL language and compiler to support Pascal.

NIDL has stronger remote procedure call semantics than XDR. For example, an RPC may be defined as idempotent. NIDL and NDR are part of HP/Apollo's Network Computing Architecture [Kong90].

NIDL/NDR have been adopted by the OSF/DCE [OSF/RPC90]. This may not be a wise decision as NDR have difficulty in handling the aggregation of variable sized types. For example, a structure can only contain one array with a variable number of elements; all components of a structure except the last one must be of fixed size. Pointers cannot be used to get around this limitation as NDR do not allow pointers. Because of the two byte length fields for strings and arrays, NDR limits the number of characters in a string and the number of array elements to less than 65,536. These limitations may lead to overly restrictive data structures. Fortunately, the OSF/DCE RPC mechanism allows negotiation of the encoding rules that are used for a connection.

2.6.2 NDR: Network Data Representation

The NDR [Zahn90] encoding rules are unique in that parts of the encoding rules are defined by the host that does the encoding. For example, if running on a Sun, integers, character strings and real numbers will be encoded in Sun's standard representation, namely big endian integers, ASCII strings and IEEE real numbers. The encoded value includes a format label that defines which type representations were used. The receiving host is responsible for converting these values to its internal representation when it decodes the value.

This technique optimizes NDR for communication between hosts with the same architecture. Thus, hosts only need one encoding routine for each primitive type but they must have different decoding routines for each type to support all of the other representations.

NIDL has twenty-five abstract types that map into nineteen NDR transfer types. In addition to these types, there is a 4 byte long format label type to define how the integers,

reals and strings were encoded.

The primitive NIDL types are:

- small - 8 bit long, 2's complement integer
- short - 16 bit long, 2's complement integer
- int - 32 bit long, 2's complement integer
- long - 32 bit long, 2's complement integer
- hyper - 64 bit long, 2's complement integer
- unsigned small - 8 bit long binary integer
- unsigned short - 16 bit long binary integer
- unsigned int - 32 bit long binary integer
- unsigned long - 32 bit long binary integer
- unsigned hyper - 64 bit long binary integer
- short enum - 16 bit long 2's complement integer
- enum - 32 bit long 2's complement integer
- byte - 8 bits long uninterpreted data
- float - 32 bits
- double - 64 bits
- char - always unsigned, ASCII or EBCDIC
- boolean - 1 byte, 0 for false, non-zero for true

- bitset enum 32 bits
- short bitset enum 16 bits
- string0 - null terminated string, ASCII or EBCDIC
- void - not encoded

The constructor NIDL types are:

- struct
- union
- array

NDR aligns values to 2, 4 or 8 byte boundaries depending on their type. This alignment technique can yield smaller encodings (fewer padding bytes) than by using XDR's 4 byte alignment.

All of the components of a `struct`, except the last one must be a fixed size. This restriction alone makes it difficult to specify some complex data structures in NIDL. There is a "transmit-as" mechanism to translate complex data structures into simpler ones that can be encoded in NDR, but it does not solve the problem very well and another layer is added to encoding and decoding.

2.7 C Based Encoding Rules

For comparison purposes, we defined C Based Encoding Rules. They show what performance can be achieved by assuming that the communicating hosts share the same architecture and that their protocol implementations use the same language and possibly compiler. These encoding rules would only be viable under these restricted conditions.

When encoding with these encoding rules, we assume that the C data structure's components have been carefully allocated contiguously from a single block of memory and that the base address of this block and its size are known. The encoding process simply traverses the C data structure in a depth first fashion and converts any non-null pointers into an offset from the start of the block. Null (zero) pointers are left as is. Since a zero offset could be a valid "address" (i.e. reference to the first component of the encoding), pointer references cannot be made to the first component in the block.

The encoding process could also be done by sending the base address of the block with the block instead of translating the pointers. The decoder would then subtract the original base address from the pointers before adding the current base address. This would eliminate the zero offset/null pointer ambiguity. We did not use this improvement in our tests.

For decoding, we assume that the encoded value is in a contiguous representation and that the base address of the encoded value's memory block is known. The decoding process simply adds the current base address to the offsets while traversing the data structure in a top down manner.

The C based encoding rules do not allow for cycles in the data structure or components to be referenced more than once by pointer as this would complicate the algorithm and provide a feature not found in the other encoding rules. This is not to say these features could be not implemented.

Chapter 3

Related Work

3.1 Performance Issues and Ideas

There is a general notion that BER encoders and decoder are slow and produce large encodings. This may be due to the richness of ASN.1 and BER or existing widely-available implementations such as ISODE. On this assumption people have devised optimizations to BER: new "light-weight" encoding rules and compaction-oriented encoding rules.

BER encodings contain explicit type information that is often unnecessary for many protocols since the receivers usually know the data types of the PDUs they receive. New encoding rules such as PER have been devised to produce very compact encodings that contain no type information.

To improve throughput, attempts have been made to minimize the cost of translation between the local representation and the transfer representation. These "light weight" encoding rules often resort to using fixed-size types that use the same representation as commonly used CPUs.

Huitema, Chave and Doghri [Huitema89] defined "Light Weight" Encoding Rules (LWER) as a faster replacement for BER. These rules used fixed-size integers and use offsets for "pointers".

Their later research [Huitema92] showed that their improved BER implementation actually had throughput similar to LWER. LWER encoded values are also considerably

larger than equivalent BER values due to the use of fixed size types. Their other benchmarks showed that BER can perform as well or better than XDR in some situations. These tests show how important implementation can be.

In optimizing their BER implementation, they found the CPU intensive areas to be memory management for decoding (35-50%) and procedure call overhead.

Fieldbus networks are designed to connect sensors, controllers and other related devices for process and industrial control applications [Thomesse87]. Fieldbus applications, due to real time constraints (bounded response times), need fast encoding rules. Compact encodings are also desirable due to the limited bandwidth in Fieldbus networks. The size of the implementation is also considered important for helping support low-cost hardware.

Fieldbus PDUs are expressed in ASN.1 to help compatibility with MMS [MMS87] but BER are deemed to be too slow. Several researchers have designed fast, Fieldbus oriented replacements for BER.

Pimentel [Pimentel88] defined a subset of BER for Fieldbus applications. Tags and lengths are often combined into a single octet and no indefinite length encodings are permitted. On average, the special encodings were 61% of the size of their BER equivalent. Although no implementation results were presented, the overall throughput is assumed to improve since smaller values can be sent faster and the assumption that smaller encodings reduce the complexity of the encoding and decoding implementations.

Cardoso and Tovar [Cardoso92] also present a set of encoding rules optimized for Fieldbus applications. These encoding rules are somewhat specialized in that they include protocol information such as PDU type (request, response, error). They support explicit and implicit type information. The authors note that handling OPTIONAL components is expensive in BER and provide a simpler but limited mechanism for handling them. No implementation benchmarks were presented.

Prasad and Gonzalo [Prasad93] found that Pimentel's encoding rules did not improve throughput significantly. They defined another set of Fieldbus oriented encoding rules similar to XDR to improve throughput while maintaining small encoded values.

In their rules, the only constructor types supported are SEQUENCE and CHOICE. OPTIONAL SEQUENCE elements are not permitted. Only UNIVERSAL tags are used and then only to determine the content of a CHOICE. All other types except strings were fixed in size.

The performance of their encoding rules and implementation were measured and compared to an ISODE based version. For larger PDUs, their values were 77% of the size of the BER equivalent and the encoders and decoders took 29% and 72%, respectively, of the time ISODE implementations used. For smaller PDUs, their encoding rules performed better.

These papers indicate that simplifying encoding and decoding of primitive types and their structuring can improve performance, but also that other implementation factors are very important. Our goal was to find out what these factors are and how to simplify their implementation. We also address how encoding rules can be designed to reduce the impact of these factors on the cost of encoding and decoding.

Chapter 4

Performance Considerations for the Snacc ASN.1 Compiler

4.1 Snacc Design Process and Goals

In this section we present the techniques the *snacc* generated encoders and decoders use to improve performance. A more in-depth presentation of using the compiler and the generated code can be found in [Sample93-1].

Snacc was designed primarily to provide high-performance encoders and decoders. We examined other ASN.1 compiler output and models to find problem areas. From these observations we designed a model for our ASN.1 compiler to use. This model was tested by implementing several encoders and decoders by hand before the compiler was written. Finally, *snacc* was written to produce encoders and decoders that used our design model.

The following general considerations guided the design of the encoder and decoder code generated by *snacc*:

- Do not make decisions at runtime that can be made at the time the ASN.1 is compiled.
- Avoid decoding tags more than once.
- Minimize memory allocations and copying.
- Streamline everything.

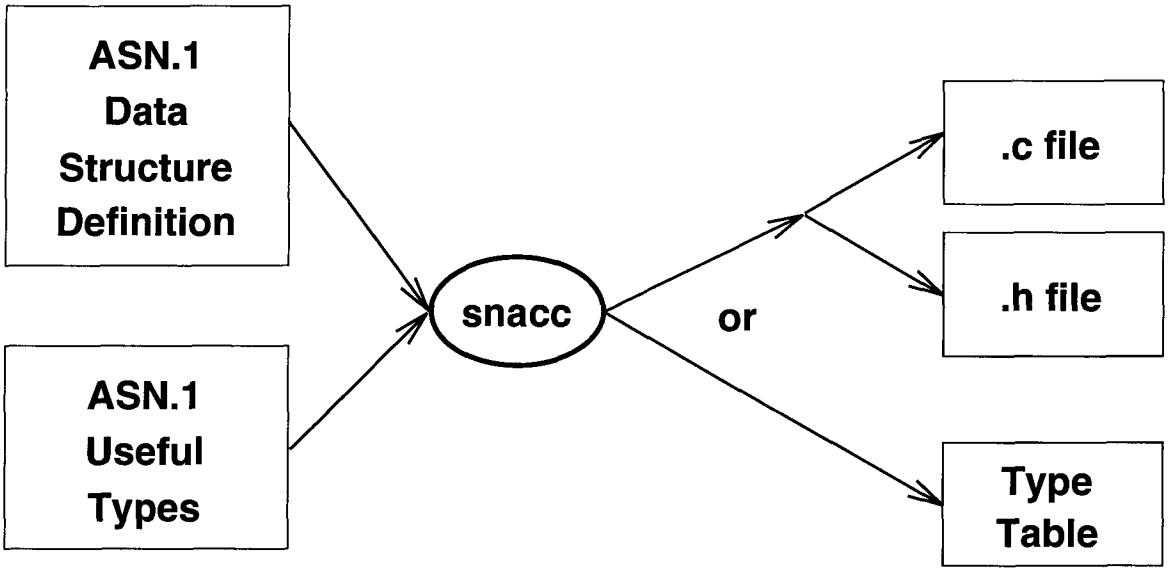


Figure 4.4: The *snacc* compiler produces C or C++ BER encode, BER decode, print and free routines or a type table.

Key areas to optimize are buffer and memory management. Buffers are used to hold encoded values and the memory management is used when building the internal representation of values when decoding. These services are provided in the library that supports the code *snacc* generates. Their interfaces were abstracted such that different implementations of the buffer and memory managers could be used without modifying the generated code. This simplifies the testing and optimizing of buffer and memory management.

Another area that caused performance problems is the handling of BER definite lengths on constructed values. The typical solution involves using an intermediate data structure between the local and transfer representations. To avoid the memory allocation and copying required by this technique, *snacc* implements backwards BER encoders.

C macros are used where possible to eliminate function call overhead for small, commonly used routines. Using macros with constant expressions as parameters allows

smarter C compilers to do some of the calculations at compile time. In general, any short-cuts that could be taken without sacrificing the robustness of code or bloating the code size too much were used. Even so, the generated code can be quite large; large reductions in the size of the binaries can be achieved by using optimizing C compilers.

Snacc behaves as follows. For each ASN.1 type an equivalent C data type, a BER encoding routine, a BER decoding routine, a printing routine and a freeing routine are generated. C values will also be generated from simple ASN.1 values. Each aspect of the C code generation will be discussed in the next sections.

Snacc can also produce type tables that allow interpreted or table based encoding and decoding. This is different from the compiled technique in that no code is generated for the ASN.1 data structures. Instead, a machine readable description of the ASN.1 data structure (defined in ASN.1 and stored in BER) is used to drive a generic encoding and decoding engine. This technique has potential code size benefits but is not designed for speed.

4.2 ASN.1 to C Data Structure Translation

Every ASN.1 type definition maps into a C typedef. SETs and SEQUENCEs map into C structures and other simple types map into their obvious C counterpart.

Memory allocations can be expensive so *snacc* makes efforts to consolidate the data structures. Small, simple types such as INTEGERs, BOOLEANs and REALs are directly included in the structs of their enclosing SEQUENCE, SET or CHOICE unless they are OPTIONAL. *Snacc* handles OPTIONAL OCTET STRING, BIT STRING and OBJECT IDENTIFIER types in a special way to reduce memory allocations and frees.

The standard technique for handling OPTIONAL SET or SEQUENCE elements is to reference the element type by pointer from the SET or SEQUENCE's C struct. If the

pointer is NULL, then the component is not present. OCTET STRINGs, BIT STRINGs and OBJECT IDENTIFIERs are included by value even when they are OPTIONAL because they are small and contain an internal pointer that can be used to determine their presence.

The SET OF and SEQUENCE OF types map into a generic list type which is doubly linked and NULL terminated. The reverse link on the lists allows for simpler backwards encoding.

4.3 Generated C Encoder Design

Snacc can generate two kinds of encoding routines for each ASN.1 type definition. One is PDU oriented and encodes the type's tag, length and content and the other only encodes the type's content. The generated encoders only call the content encoders, except in the case of ANY and ANY DEFINED BY types where tagging information is not known in the calling routine.

The most interesting thing about *snacc* generated encoders is the fact that they encode backwards [Steedman90]. Instead of writing the first component of an encoding and appending the rest in order, *snacc* encoders encode the last component and prepend the others in order.

This "backwards" encoding technique simplifies the use of definite lengths on constructed values. Other encoders that encode forwards, such as those of CASN1 [Yang88] [Neufeld90], use an intermediate buffer format so that a buffer containing the encoded length of a constructed value can be inserted before its encoded content, after the content has been encoded. Use of intermediate representations can hurt performance. Other compilers' approaches have been to only encode the indefinite length alternative for constructed values, however, this will not support some encoding rules such Distinguished

Encoding Rules (DER) [CDER92] that require definite lengths.

The drawback of encoding backwards is that BER values cannot be written to stream oriented connections as they are encoded. This removes the possibility of pipelining the encode and transmission tasks.

Both definite and indefinite length encodings for constructed values' lengths are supported. The choice is made when compiling the generated code.

Handling tags for BER encoders and decoders is bothersome. The shared nature of tag information between the enclosing type and the type itself is the root of the problem[1]. Encoding tags is a common operation for BER encoders so it must be optimized.

For example, a SET may override the UNIVERSAL tag on an OCTET STRING component by using IMPLICIT tagging. The SET controls the class and code of the tag but the OCTET STRING itself handles the form of the tag. A typical implementation would have the SET pass the tag's class and code as parameters to the OCTET STRINGs encoding routine.

For all types except OCTET STRINGs, BIT STRINGs and those derived from them, their tag form is known when the ASN.1 is compiled. OCTET STRING and BIT STRING types are primitive types that can have "constructed" encodings so their tag form can be either primitive or constructed. BER allows encoder implementations to choose between the constructed and primitive forms. To avoid an encode time decision to determine the form, *snacc* supports only the non-constructed encoding of these types. *Snacc* decoders will decode the constructed string types so BER conformance is not sacrificed.

With the above restrictions, *snacc* can calculate the exact encoding of every tag at ASN.1 compile time. To "encode" a tag, *snacc* generated encoders simply copy octet(s)

[1]The tag class and code can be defined by the enclosing type (via implicit tagging) while the type itself defines the tag's form bit.

to the buffer; the tag octet(s) are constants and require no encode time calculations.

4.4 Generated C Decoder Design

Snacc uses special techniques for representing and handling tags and provides a light-weight memory manager to optimize the decoding process. As with encoding routines, *snacc* can generate two kinds of decoding routines for each ASN.1 type definition. One is PDU oriented and the other only decodes the type's content.

Tag representation is very important for decoding. Tag comparison is a common decoding operation; it is used for determining which type is next or to verify that a particular type is present. To simplify tag comparison and passing tags as parameters, *snacc* uses `long ints` to hold tag values.

To streamline the decoding of tags, the encoded tag is copied directly into a `long int`. The first octet of the tag is placed in the high byte of the `long int` with the rest following. If a tag is only one octet long, the remaining bytes of the `long int` are zeroed. This technique imposes a restriction on the maximum tag code that can be handled. For a system with four byte `long ints`, the maximum tag code that can be represented is 2^{21}; this is far higher than can realistically be expected.

The shared nature of tag information can lead to tags being decoded more than once in some implementations. For example, due to the unordered nature of SETs, SET decoders need to decode the next tag to determine which component is to be decoded. The CASN1 decoders decode the tag once in the SET decoder routine and again in the component's decoding routine. This technique hurts performances and requires that the value being decoded be held in a buffer that allows "backing up" so the tag can be decoded again.

Snacc avoids decoding tags twice in these situations by passing the decoded tag in

its `long int` form into every content decoding routine. All types have the tag and length pairs on the content decoded by their enclosing type (e.g. SET), and the last pair is passed to the content decoding routine. This allows the enclosing type and the component type to use the tag information without decoding it twice[2].

To support this technique, CHOICE types are a special case. If a type such as a SET encloses an untagged CHOICE type it must decode the first tag in the CHOICE's content. To handle this uniformly for tagged and untagged CHOICEs, the enclosing type always decodes all of the tag and length pairs until the first tag and length of the CHOICE content is decoded. Thus, the CHOICE decoding routine expects to be passed the tag that determines which element is present in its value.

During the decoding process, memory is allocated to hold the local representation of the value. Complex data structures can require many allocations and after the value has been processed it is usually freed. *Snacc* generated decoders call the memory manager in such a way that different memory management can easily be used. Section 4.8 describes this in further detail.

Some decoder implementations use a special stack to hold tag and length information while decoding. This stack must be large enough or be able to dynamically grow to handle some large complex values. To eliminate the need for this stack and perhaps improve memory locality, *snacc* uses local variables in each generated decode routine for the tags and lengths. This technique may also make the code easier to understand for users.

Snacc decoders ignore redundant length information. Explicit tagging in ASN.1 definitions yields encodings that have extra tag and length pairs before a value. *Snacc* decoders only verify that the innermost length is correct (definite length cases only).

[2]Passing tag values on the stack is less costly than the CASN1 technique of decoding twice which calls an extra routine. Only OCTET STRING, BIT STRING and CHOICE decoding routines will actually use the tag parameter. Optimizing compilers can eliminate it for the routines that do not use it.

To save memory, decoders generated by some other tools build values that reference the data in the encoded PDU for types like OCTET STRING. *Snacc* decoded values do not reference the BER data in any way for several reasons. One, the encoded value may be held in some bizarre buffer making access to the value difficult. Two, with more encoding rules being formalized, this technique may not always work since the encoded format may be different from the desired internal format. Three, *snacc* decoders concatenate any constructed BIT and OCTET STRINGs values when decoding, to simplify processing in the application. The penalty for this is an extra copy for the string type components.

Some protocol implementations may benefit from directly referencing data in the PDU buffer from the local representation. For example, the performance of an X.400[3] email implementation that receives PDUs from the transport layer in a single memory block could save a copy by referencing the data directly. However, if UNALIGNED PER were used to encode the X.400 PDU, the referenced string may not be octet aligned, which limits the benefit of this technique.

4.5 Error Management

Error management is important but it must be implemented without incurring too much cost for the common case where there are no errors. One method of streamlining error management is to eliminate it by assuming that the decoded values are always correct. This is not acceptable for a reliable implementation.

Snacc decoders can detect a variety of errors. All tagging errors are reported. SETs must contain all non-OPTIONAL components and SEQUENCEs must be in order and contain all non-OPTIONAL components. Extra components in SETs and SEQUENCEs are considered an error. Errors will also be reported if you attempt to decode values that

[3]X.400 PDUs are dominated by a large string body part.

exceed the limitations of the internal representation (e.g. an integer that is larger than a `long int` allows).

Snacc encoders only do minimal error checking; they assume that the value to be encoded is well formed. The buffer writing routines called by the encoders will set a flag if the buffer overflows or no memory is available to extend the buffer. After the buffer write error flag is set, future buffer writes are treated as no-ops. When the encoding routine returns, the user can check the status of this flag to ensure that the value was encoded properly.

The decoding process is much more prone to errors. The value being decoded may have errors from network transmission or a faulty encoder. Local resource errors, such as insufficient memory, can also cause decoding errors.

To implement a complete yet light-weight error management scheme for the decoders, we used the `setjmp` and `longjmp` functions [Kernighan88]. Previously, the availability of these functions was very system dependent, however, they are now part of the ANSI Standard C Library.

Before decoding begins, *setjmp* is called to record the current environment (processor registers etc.). The environment value set by *setjmp* is then passed into the decoding routine. Every decoding routine takes this parameter.

When the decoding routines encounter a serious error such as running out of memory for the decoded value, they call `longjmp` with the environment value they were given as a parameter, along with a unique error code. This "returns" execution directly to where *setjmp* was called along with the error code.

The *setjmp* and `longjmp` based error management is simple and does not impact the performance of decoding correctly encoded values (other than an extra parameter to each decoding routine). Other error management techniques such as passing back error codes that the calling functions must check will affect the decoding performance even for

```
jmp_buf env;
...
if ((val = setjmp(env)) == 0)
   BDecPersonnelRecord( &buf, &pr, &decodedLen, env);
else
{
    decodeErr = TRUE;
    fprintf(stderr,"ERROR - Decode routines returned %d\n",val);
}
```

Figure 4.5: The `setjmp` function is called before decoding

```
if (mandatoryElmtCount1 != 2)
{
    Asn1Error("BDecChildInformationContent: ERROR - non-optional elmt\
missing from SET.\n");
    longjmp(env, -108);
}
```

Figure 4.6: The `longjmp` function is called to signal an error

correctly encoded values.

Figures 4.5 and 4.6 show code fragments that illustrate how *snacc* decoders use `setjmp` and `longjmp` for error handling.

4.6 Compiler Directives

Snacc attempts to produce a C data structure definition that is compact while not too restrictive. However, users can often make optimizations that *snacc* cannot. To allow this form of optimization, *snacc* provides compiler directives that allow users to influence the generated C data structure and the encoding and decoding routines. These can be used to consolidate the data structure (by removing pointer references thus saving memory allocations) or to replace the encode and decode routines with more efficient ones that

are tailored to the target environment.

The compiler directives have the form:

```
--snacc <attribute>:"<value>"  <attribute>:"<value>" ...
```

The `attribute` is the name of one of the *snacc* defined attributes and the `value` is what the `attribute`'s new value will be. The attribute value pairs can be listed in a single `--snacc` comment or spread out is a list of consecutive comments. Since these directives are in ASN.1 comments, they do not make the ASN.1 unusable by other ASN.1 tools.

Compiler directives are only accepted in certain places in the ASN.1 code. Depending on their location, the compiler directives affect type definitions or type references. The directives for type definitions and references are different.

The following example illustrates their use. Assume a data structure always deals with `PrintableStrings` that are null terminated. The default *snacc* string type is a structure that includes a length and a pointer to the string's octets. To change the default type to a simple pointer (`char*`) the best way would be define a new string type, `MyString` as follows:

```
Foo ::= SET
{
    s1 [0] MyString OPTIONAL,
    s2 [1] MyString,
    i1 [2] INTEGER
}

Bar ::= CHOICE
{
    s1 MyString,
    i1 INTEGER
}
```

```
Bell ::= MyString

MyString ::= --snacc isPtrForTypeDef:"FALSE"
             --snacc isPtrForTypeRef:"FALSE"
             --snacc isPtrInChoice:"FALSE"
             --snacc isPtrForOpt:"FALSE"
             --snacc optTestRoutineName:"MYSTRING_NON_NULL"
             --snacc genPrintRoutine:"FALSE"
             --snacc genEncodeRoutine:"FALSE"
             --snacc genDecodeRoutine:"FALSE"
             --snacc genFreeRoutine:"FALSE"
             --snacc printRoutineName:"printMyString"
             --snacc encodeRoutineName:"EncMyString"
             --snacc decodeRoutineName:"DecMyString"
             --snacc freeRoutineName:"FreeMyString"
             PrintableString --snacc cTypeName:"char*"
```

All but the last `--snacc` comment bind with the `MyString` type definition. The last directive comment binds with the `PrintableString` type. The C data structure resulting from the above ASN.1 and *snacc* compiler directives is the following:

```
typedef char* MyString; /* PrintableString */

typedef struct Foo /* SET */
{
    MyString s1; /* [0] MyString OPTIONAL */
    MyString s2; /* [1] MyString */
    AsnInt i1; /* [2] INTEGER */
} Foo;

typedef struct Bar /* CHOICE */
{
    enum BarChoiceId
    {
        BAR_S1,
        BAR_I1
    } choiceId;
    union BarChoiceUnion
    {
    MyString s1; /* MyString */
```

```
    AsnInt i1; /* INTEGER */
    } a;
} Bar;

typedef MyString Bell; /* MyString */
```

The compiler directives used on the `MyString` type have some interesting effects. Notice that `MyString` is not referenced by pointer in the CHOICE, SET, or type definition, `Bell`. The *snacc* compiler directives are described fully in [Sample93-1].

4.7 Buffer Management

Encoding and decoding performance is greatly affected by the cost of writing to and reading from buffers. Thus, efficient buffer management is necessary. Flexibility is also important to allow integration of the generated encoders and decoders into existing environments. To provide both of these features, the calls to the buffer routines are actually macros that can be configured as desired. They must be configured prior to compiling the encode/decode library and the generated code. Since virtually all buffer calls are made from the encode/decode library routines, buffer routine macros should not bloat code size significantly.

If a single, simple buffer type can be used in the target environment, the buffer routine macros can be defined to call the macros for the simple buffer type. This results in the buffer type being bound at the time the generated code is compiled, with no function call overhead from the encode or decode routines. However, this also means that the encode and decode library will only work with that buffer type.

If multiple buffer formats must be supported at runtime, the buffer macros can be defined like the *ISODE* buffer calls, where a buffer type contains pointers to the buffer routines and data of user defined buffer type. This approach will hurt performance since each buffer operation will be an indirect function call.

The backwards encoding technique used by *snacc* requires special buffer primitives that write from the end of the buffer towards the front. The decoder routines require forward buffer reading routines.

The encode and decode library routines deal with buffers via the `BUF_TYPE` type and nine buffer routines with the following names and prototypes:

- `unsigned char BufGetByte(BUF_TYPE b);`
- `unsigned char BufPeekByte(BUF_TYPE b);`
- `char* BufGetSeg(BUF_TYPE b, unsigned long int* lenPtr);`
- `void BufCopy(char* dst, BUF_TYPE b, unsigned long int* lenPtr);`
- `void BufSkip( BUF_TYPE b, unsigned long int len);`
- `void BufPutByteRvs( BUF_TYPE b,  unsigned char byte);`
- `void BufPutSegRvs( BUF_TYPE b, char* data, unsigned long int len);`
- `int BufReadError(BUF_TYPE b);`
- `int BufWriteError(BUF_TYPE b);`

To map the library's calls to buffer routines to your own buffer routines, a configuration header file is used. For example, to configure *snacc* to use the buffer type `MyBuffer`, the following would appear in the configuration file.

```
#include "my-buf.h"
#define BUF_TYPE MyBuffer**
#define BufGetByte(b)              MyBufGetByte(b)
#define BufGetSeg(b, lenPtr)       MyBufGetSeg(b, lenPtr)
#define BufCopy( dst, b, lenPtr)   MyBufCopy( dst, b, lenPtr)
#define BufSkip( b, len)           MyBufSkip(b, len)
#define BufPeekByte( b)            MyBufPeekByte( b)
```

```
#define BufPutByteRv( b,  byte)     MyBufPutByteRv( b, byte)
#define BufPutSegRv( b, data, len) MyBufPutSegRv( b, data, len)
#define BufReadError(b)             MyBufReadError(b)
#define BufWriteError(b)            MyBufWriteError(b)
```

4.8 Memory Management

Like buffer management, memory management is very important for efficient decoders. *Snacc* decoders allocate memory to hold the decoded value. After the decoded value has been processed it is usually freed. The decoding and freeing routines in the library and the ones generated by *snacc* both use the memory manager. The decoders allocate memory with the `Asn1Alloc` routine and the freeing routines call `Asn1Free`.

The configuration header file allows the user to change the default memory manager prior to compiling the library and the generated code. *Snacc* provides a particularly efficient memory manager, discussed shortly, called "nibble memory".

The memory manager must provide three routines: `Asn1Alloc`, `Asn1Free` and `CheckAsn1Alloc`. These memory routines should have the following interfaces:

```
void* Asn1Alloc(unsigned long int size);
void  Asn1Free(void* ptr);
int   CheckAsn1Alloc(void* ptr, ENV_TYPE env);
```

The decoders assume that `Asn1Alloc` returns a zeroed block of memory. This saves explicit initialization of OPTIONAL elements with NULL in the generated decoders. The `ENV_TYPE` parameter is used with the error management system for calls to `longjmp`.

To change the memory management system the configuration file needs to be edited. For example, if performance is not an issue and you want to use `calloc` and `free`, the configuration file would be as follows:

```
#include "malloc.h"
#define Asn1Alloc(size)  calloc(1, size)
```

```
#define Asn1Free(ptr)    free(ptr)
#define CheckAsn1Alloc(ptr, env)\
   if ((ptr) == NULL)\
      longjmp(env, -27);
```

The nibble memory system does not need explicit frees of each component so the generated free routines are not needed. However, if you change the memory management to use something like `malloc` and `free` you should use the generated free routines.

By default, *snacc* uses a nibble memory scheme to provide efficient memory management. Nibble memory works by getting a large block of memory from the O/S for allocating smaller requests. When the decoded PDU is no longer required by the application the entire space allocated to the PDU can be freed by calling a routine that simply resets a pointer. There is no need to use the *snacc* generated free routines to traverse the entire data structure, freeing a piece at a time.

If `calloc` and `free` are used for memory management instead of nibble memory, the generated free routines must be used to free the values. The generated free routines hierarchically free all of a PDU's memory using a depth first algorithm.

4.9 Novel Non-performance Related Features

In this section, the method *snacc* uses to handle the encoding and decoding of ANY DEFINED BY types is described.

The ANY and ANY DEFINED BY type are classically the most irritating ASN.1 types for compiler writers. They rely on mechanisms outside of ASN.1 to specify what types they contain. The 1992 ASN.1 standard has rectified this by adding much stronger typing semantics and eliminating macros.

The ANY DEFINED BY type can be handled automatically by *snacc* if the SNMP OBJECT-TYPE [snmp90] macro is used to specify the identifier value to type mappings.

The identifier can be an INTEGER or OBJECT IDENTIFIER. Handling ANY types properly will require modifications to the generated code since there is no identifier associated with the type.

The general approach used by *snacc* to handle ANY DEFINED BY types is to lookup the identifier value in a hash table for the identified type. The hash table entry contains information about the type such as the routines to use for encoding and decoding.

Two hash tables are used, one for INTEGER to type mappings and the other for OBJECT IDENTIFIER to type mappings. *Snacc* generates an `InitAny` routine for each module that uses the OBJECT-TYPE macro. This routine adds entries to the hash table(s). The `InitAny` routine(s) is called once before any encoding or decoding is done.

The hash tables are constructed such that an INTEGER or OBJECT IDENTIFIER value will hash to an entry that contains:

- the `anyId`
- the INTEGER or OBJECT IDENTIFIER that maps to it
- the size in bytes of the identified data type
- a pointer to the type's PDU encode routine
- a pointer to the type's PDU decode routine
- a pointer to the type's print routine
- a pointer to the type's free routine

The referenced encode and decode routines are PDU oriented in that they encode the type's tag(s) and length(s) as well as the type's content.

Snacc builds an `enum` called `AnyId` that enumerates each mapping defined by the OBJECT-TYPE macros. The name of the value associated with each macro is used

as part of the enumerated identifier. The `anyId` in the hash table holds the identified type's `AnyId enum` value. The `anyId` is handy for making decisions based on the received identifier, without comparing OBJECT IDENTIFIERs. If the identifiers are INTEGERs then the `anyId` is less useful.

With ANY DEFINED BY types, it is important to have the identifier decoded before the ANY DEFINED BY type is decoded. Hence, an ANY DEFINED BY type should not be declared before its identifier in a SET since SETs are un-ordered. An ANY DEFINED BY type should not be declared after its identifier in a SEQUENCE. *Snacc* will print a warning if either of these situations occur.

The hash tables may be useful to plain ANY types which do not have an identifier field like the ANY DEFINED BY types; the OBJECT-TYPE macro can be used to define the mappings and the `SetAnyTypeByInt` or `SetAnyTypeByOid` routine can be called with the appropriate identifier value before encoding or decoding an ANY value. The compiler will insert calls to these routines where necessary with some of the arguments left as "???". There will usually be a "`/* ANY - Fix me ! */`" comment before code that needs to be modified to correctly handle the ANY type. The code generated from an ASN.1 module that uses the ANY type will not compile without modifications.

OPTIONAL ANYs and ANY DEFINED BY types that have not been tagged are a special problem for *snacc*. Unless they are the last element of a SET or SEQUENCE, the generated code will need to be modified. *Snacc* will print a warning message when it encounters one of these cases.

To illustrate how ANY DEFINED BY values are handled, we present typical encoding and decoding scenarios. Each ANY or ANY DEFINED BY type is represented in C by the `AsnAny` type which contains only a `void*` named `value` to hold a pointer to the value and a `AnyInfo*` named `ai` which points to a hash table entry.

When encoding, before the ANY DEFINED BY value is encoded, `SetAnyTypeByOid`

or `SetAnyTypeByInt` (depending on the type of the identifier) is called with the current identifier value to set the `AsnAny` value's `ai` pointer to the proper hash table entry. Then to encode the ANY DEFINED BY value, the encode routine pointed to from the hash table entry is called with the `value void*` from the `AsnAny` value. The `value void*` in the `AsnAny` should point to a value of the correct type for the given identifier, if the user set it up correctly. Note that setting the `void*` value is not type safe; one must make sure that the value's type is the same as indicated by the identifier.

For decoding, the identifier must be decoded prior to the ANY DEFINED BY value otherwise the identifier will contain an uninitialized value. Before the ANY or ANY DEFINED BY value is decoded, `SetAnyTypeByOid` or `SetAnyTypeByInt` (depending on the type of the identifier) is called to set the `AsnAny` value's `ai` pointer to the proper hash table entry. Then a block of memory of the size indicated in the hash table entry is allocated, and its pointer stored in the `AsnAny` value's `void*` entry. Then the decode routine pointed to from the hash table entry is called with the newly allocated block as its value pointer parameter. The decode routine fills in the value assuming it is of the correct type.

There is a problem with *snacc*'s method for handling ANY DEFINED BY types for specifications that have two or more ANY DEFINED BY types that share some identifier values. Since only two hash tables are used and they are referenced using the identifier value as a key, duplicate identifiers will cause unresolvable hash collisions.

4.10 C++ Design

The C++ backend of *snacc* was designed after the C backend had been written. The basic model that the generated C++ uses is similar to that of the generated C, but benefits from the object oriented features of C++.

Some cleaner designs were rejected either due to their poor performance or the inability of the available C++ compiler to handle certain language features.

Tags and lengths would fit nicely into their own classes but performance was considerably worse than the technique used in the C model. The C design was retained in the C++ model for this reason.

For error management, C++'s `try` and `catch` [Stroustrup91] are obvious replacements for the `setjmp` and `longjmp` used by the C decoders. Unfortunately, this is a newer C++ feature and was not yet supported.

C++ templates are very attractive for type safe lists (for SET OF and SEQUENCE OF) without duplicating code. Since template support was weak in the available C++ compiler, templates were rejected. Instead, each SET OF and SEQUENCE OF type generates its own new class with all of the standard list routines.
Every ASN.1 type is represented by a C++ class with the following characteristics:

1. inherits from the `AsnType` base class
2. has a parameterless constructor
3. has a clone method, `Clone`
4. has a PDU encode and decode method, `BEnc` and `BDec`
5. has a content encode and decode method, `BEncContent` and `BDecContent`
6. has top level interfaces to the PDU encode and decode methods (handles the `setjmp` etc.) for the user, `BEncPdu` and `BDecPdu`
7. has print method, `Print`

The following C++ fragment shows the class features listed above in greater detail:

```
class Foo : public AsnType
{
   ... // data members

  public:
    Foo();  // requires parameterless constructor

    // PDU (tags/lengths/content) encode and decode routines
    AsnLen BEnc(BUF_TYPE b);
    void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env);

    // content encode and decode routines
    AsnLen BEncContent(BUF_TYPE b);
    void   BDecContent(BUF_TYPE b, AsnTag tag, AsnLen elmtLen,
                       AsnLen& bytesDecoded, ENV_TYPE env);

    // methods most likely to be used by your code.
    // Returns non-zero for success
    int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded);
    int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded);

    void Print(ostream& os);
    AsnType* Clone();
}
```

In brief, the `AsnType` provides a base class that has virtual `BEnc`, `BDec` and `Clone` routines. The purpose of this class is to provide a generic ASN.1 type class that can be used to handle ANY and ANY DEFINED BY types. The `Clone`[4] routine is used to generate a new instance (not a copy) of the object by calling the constructor of the object that it is invoked on. This allows the ANY DEFINED BY type decoder to create a new, initialized object of the correct type from one stored in a hash table when decoding. The virtual `BDec` method of the newly created object is called from `AsnAny` class's `BDec` method. When encoding, the `AsnAny` objects call the virtual `BDec` method of the type they contain.

[4] `Clone` is a poorly chosen name. `NewInstance` would be better.

4.11 Table Driven Encoding and Decoding

Snacc can generated type tables as well as source code. Type tables can be used instead of generated routines for encoding and decoding. A type table can be loaded during runtime and table routines from the library can use the ASN.1 type information in the type table to encode, decode and for other things. The current table routines only support C.

No C code specific to the ASN.1 data structure in a type table is used by the table routines. When decoding using a type table, the decoding routine builds a data structure for the local representation using some simple rules.

The rules for building a C data structure during decoding are as follows. For SEQUENCEs and SETs the decoder will allocate an array of `void*` types. The array has one entry for each SET component. Each entry in the array points to a value of the expected type.

The standard *snacc* library types are used for the primitive and list types. CHOICE types are similar to the ones *snacc* generates except the `union` is replaced with a `void*` since no type checking is done. CHOICEs are represented by a `struct` with a `long int` to define the element that is present and `void*` to point to the component of the CHOICE that is present.

The table driven encoder expects values passed to it for encoding to be of the same structure.

Dealing with a complex data structure is difficult without a C definition for it. Type tables can be used to generate a header file that contains a friendly, properly typed version of the data structure that the table routines will use for the given type table. The table routines do not use this header file but the user can in his code.

Appendix C contains the ASN.1 definition of the type tables *snacc* generates. The type table data structure was defined in ASN.1 so that type tables are simple to encode

to disk by using the snacc generated encoding routines. Conceivably, these type tables could be sent over a network to dynamically configure a network device or to configure network management and protocol testing tools.

Chapter 5

Implementation of the Snacc Compiler

5.1 Snacc Compiler Design

The *snacc* compiler is implemented with *yacc*, *lex* (actually GNU's equivalents, *bison* and *flex*), and C. Despite the shortcomings of *lex* and *yacc*, they provide reasonable performance without too much programming effort. Since *yacc* parsers are extremely difficult to modify during runtime, any macro that the compiler is to handle must be hand coded into the ASN.1 *yacc* grammar followed by recompilation of *snacc*. Macro definitions do not need special consideration since they are skipped by the compiler. Macro definitions and complex value notation are kept as text in the data structure resulting from a parse. This text can be parsed (via modifying the compiler) if desired.

The syntax of the ASN.1 language was not initially designed with machine processing in mind. The ASN.1 grammar contains several ambiguities, and features such as macros allow users to change the grammar. The fact that types can be referenced before they are defined complicates parsing.

To handle the anti-compiler nature of ASN.1's syntax, *snacc* makes several passes on the parse tree data structure when compiling. The term "pass" is a bit misleading as none of these traversals creates temporary files; this allows *snacc* to process large ASN.1 specifications quite quickly. The compiler passes are explained in the next sections. The main passes of the compiler are executed in the following order:

1. Parse useful types ASN.1 module.

2. Parse all user specified ASN.1 modules.
3. Link local and imported type references in all modules.
4. Parse values in all modules.
5. Link local and imported value references in all modules.
6. Process any macro types.
7. Normalize types.
8. Mark recursive types and signal any recursion related errors.
9. Check for semantic errors in all modules.
10. Generate C/C++ type information for each ASN.1 type.
11. Sort the types from least dependent to most dependent.
12. Generate the C/C++ encoders and decoders.

5.2 Pass 1: Parsing the Useful Types Module

The ASN.1 useful types are not hardwired into *snacc*. Instead they have been placed in a separate ASN.1 module. This allows the user to define his own useful types or re-define the existing ones without modifying *snacc* . This also has the benefit that names of useful types are not key words in the lexical analyzer. This step is not really a compiler pass on the module data, however it is described as one for simplicity.

Snacc handles the useful types module differently from the other modules. Instead of parsing the module and generating code for it, *snacc* parses the module and makes the types in it accessible to all of the other modules being parsed. Note that the other

modules do not need to explicitly import from the useful types module. See Section 5.4 for more information on how useful types affect linking.

The encode, decode and other routines for the useful types are in the runtime library. Currently, the useful types library routines are the same as the ones the compiler would normally generate given the useful types module. However, since they are in the library, they can be modified. For example, one could check character sets (string types), or use local time formats to represent their BER equivalent (UTCTime, GeneralizedTime). The following types are in the useful types module:

```
ASN-USEFUL DEFINITIONS ::=
BEGIN
ObjectDescriptor ::= [UNIVERSAL 7]  IMPLICIT OCTET STRING
NumericString    ::= [UNIVERSAL 18] IMPLICIT OCTET STRING
PrintableString  ::= [UNIVERSAL 19] IMPLICIT OCTET STRING
TeletexString    ::= [UNIVERSAL 20] IMPLICIT OCTET STRING
T61String        ::= [UNIVERSAL 20] IMPLICIT OCTET STRING
VideotexString   ::= [UNIVERSAL 21] IMPLICIT OCTET STRING
IA5String        ::= [UNIVERSAL 22] IMPLICIT OCTET STRING
GraphicString    ::= [UNIVERSAL 25] IMPLICIT OCTET STRING
VisibleString    ::= [UNIVERSAL 26] IMPLICIT OCTET STRING
ISO646String     ::= [UNIVERSAL 26] IMPLICIT OCTET STRING
GeneralString    ::= [UNIVERSAL 27] IMPLICIT OCTET STRING
UTCTime          ::= [UNIVERSAL 23] IMPLICIT OCTET STRING
GeneralizedTime  ::= [UNIVERSAL 24] IMPLICIT OCTET STRING

EXTERNAL         ::= [UNIVERSAL 8] IMPLICIT SEQUENCE
{
    direct-reference      OBJECT IDENTIFIER OPTIONAL,
    indirect-reference    INTEGER OPTIONAL,
    data-value-descriptor ObjectDescriptor OPTIONAL,
    encoding CHOICE
    {
        single-ASN1-type [0] OCTET STRING,  -- should be ANY
        octet-aligned    [1] IMPLICIT OCTET STRING,
        arbitrary        [2] IMPLICIT BIT STRING
    }
}
END
```

5.3 Pass 2: Parsing ASN.1 Modules

During this pass, all of the specified modules are parsed into the *Module* data structure. The ASN.1 source files are not consulted again. *Yacc* and *lex* are doing the work in this step.

A lexical tie-in is where the *yacc* parser puts the lexical analyzer into a different mode (and is usually considered a hack). The different modes tokenize symbols differently, which is useful for skipping well delimited sections that cannot be parsed easily by a *yacc* parser on the first pass. Lexical tie-ins are used in two places to simplify the ASN.1 grammar sufficiently for *yacc* and *lex*. There are two special modes in the lexical analyzer, one for ASN.1 macro definitions and the other for ASN.1 values enclosed in {}'s.

The lexical tie-in for scanning macro definition bodies works with macro definitions of the following form:

```
<upper case identifier> MACRO ::= BEGIN ... END
```

Everything between the BEGIN and END is stuffed into a string by *lex* and passed back as single token to the *yacc* parser.

Values within {}'s are parsed in a similar way. Value parsing cannot really be done at this stage since complete type information is needed and the types are not fully parsed or linked yet.

Most syntax errors are reported during this pass. If syntax errors are encountered, *snacc* will report as many as it can from the offending module before the parser is hopelessly lost. If the types and values are separated with semi-colons, the parser can recover after a syntax error and attempt to find more errors in that module before exiting.

5.4 Pass 3: Linking Types

The third pass links all type references [1]. *Snacc* attempts to resolve any currently visible (i. e. not in macro definitions or constructed values) type reference. This includes type references in simple value definitions and subtyping information. The useful types module (if given) is linked first.

Snacc will exit after this pass if any type references could not be resolved. Error messages with file and line number information will be printed to *stderr*.

First, each module identifier is checked for conflicts with the others. If the module identifier includes an OBJECT IDENTIFIER, *snacc* only checks for conflicts with the other module identifier OBJECT IDENTIFIERs. When only a module name is provided, *snacc* checks for conflicts with the other module names, even if the other module identifiers include OBJECT IDENTIFIERs. If the OBJECT IDENTIFIER of a module identifier contains any value references, it will be ignored for module look-up purposes. Note that value references within the module identifier OBJECT IDENTIFIERs are not allowed in the 1992 version of ASN.1 due to the difficulty in module name resolution.

Two modules with the same name but different OBJECT IDENTIFIERs are not considered an error within ASN.1. However, because the generated files use the module name as part of their name, the code generation pass will fail.

Next, each module's import lists are resolved by finding the named modules and then verifying that the named module contains all of the types imported from it. Then for each module, each type reference (except those of the form *modulename.typename*) is assumed to be a local type reference and the linker attempts to find a local type definition of the same name. If a matching local definition is found, the type reference is resolved and the linker continues with the next type reference.

[1] This pass also counts and stores the number of times a type definition is referenced locally and from other modules. This information is used during the type sorting pass.

For each type reference of the form *modulename.typename*, the linker looks in the module with name *modulename* for the type *typename*. If the type is found the reference is resolved, otherwise a linking error is reported. Note that this form of type reference provides a special scope that does not conflict with other local or imported types in that module. For type references that failed to resolve locally and are not of the form *modulename.typename*, the linker looks in the import lists of the current type reference's module for a type to resolve with. If the type is found in the import lists, the reference is resolved.

For the remaining unresolved type references (failed local and legal import resolution and are not of the form *modulename.typename*), the linker looks in the useful types module. If the type is found in the useful types module then the reference is resolved, otherwise a linking error is reported. Note that when a useful types module is specified, it is globally available to all modules, but it has the lowest linking priority. That is, if a type reference can be resolved legally without the useful types module, it will be.

Some type checking must be done in this pass to link certain types properly. These include:

- a SELECTION type must reference a field of a CHOICE type.
- a COMPONENTS OF type in a SET must reference a SET.
- a COMPONENTS OF type in a SEQUENCE must reference a SEQUENCE.

5.5 Pass 4: Parsing Values

The fourth pass attempts to parse any value that is enclosed in {}'s. INTEGERS, REALs and BOOLEANS that are not enclosed in braces are parsed in the first pass. The value parser uses each value's type information to help parse the value. Values within {}'s

hidden within types such as default values and parts of subtypes are not parsed. Since subtypes and default values do not affect the generated code, upgrading the value parser in this respect is not very useful.

The only type of value in {}'s that is parsed is the OBJECT IDENTIFIER. All of the OBJECT IDENTIFIER value forms are supported but *snacc* loosens the restrictions on using arc names defined in the OBJECT IDENTIFIER tree. ASN.1 allows OBJECT IDENTIFIER values to reference special built-in arc names from the OBJECT IDENTIFIER tree defined in Annexes B, C and D of CCITT X.208. For example the first arc in an OBJECT IDENTIFIER value can be either *ccitt*, *iso* or *joint-iso-ccitt*. The acceptable arc names are context dependent; for example the second arc can be one of *standard*, *registration-authority*, *member-body* or *identified-organization* only if the first arc was *iso* or 1.

Snacc uses a simplified algorithm to handle references to the arc names defined in the OBJECT IDENTIFIER tree. Any arc value that is represented by a single identifier is checked to see if it is one of the arc names defined in the OBJECT IDENTIFIER tree; context is ignored. If the identifier matches one of the arc names then its value is set accordingly. The lack of context sensitivity in *snacc*'s algorithm may cause the arc name to link with an arc name from the OBJECT IDENTIFIER tree when a local or imported INTEGER was desired. If this happens, *snacc* will produce an erroneous C/C++ value definition for that OBJECT IDENTIFIER. The following is the list of special arc names that *snacc* understands.

- ccitt = 0
- iso = 1
- joint-iso-ccitt = 2
- standard = 0

- registration-authority = 1
- member-body = 2
- identified-organization = 3
- recommendation = 0
- question = 1
- administration = 2
- network-operator = 3

5.6 Pass 5: Linking Values

The fifth pass links value references. The value linker looks for value references to resolve in value definitions and type definitions, including default values and subtyping information. The value linking algorithm is virtually identical to the type linking pass (see Section 5.4).

Currently the value parsing is limited to OBJECT IDENTIFIER values. Simple values that are not between {}'s are parsed in the first pass. Here is an example that illustrates the OBJECT IDENTIFIER parsing and linking. The following values:

```
foo OBJECT IDENTIFIER ::= { joint-iso-ccitt 2 88 28 }
bar OBJECT IDENTIFIER ::= { foo 1 }
bell INTEGER ::= 2
gumby OBJECT IDENTIFIER ::= { foo bell }
pokie OBJECT IDENTIFIER ::= { foo stimpy(3)}
```

are equivalent to this:

```
foo OBJECT IDENTIFIER ::= { 2 2 88 28 }
```

```
bar OBJECT IDENTIFIER ::= { 2 2 88 28  1 }
bell INTEGER ::= 2
gumby OBJECT IDENTIFIER ::= { 2 2 88 28 2}
pokie OBJECT IDENTIFIER ::= { 2 2 88 28 3}
```

5.7 Pass 6: Processing Macros

The fifth pass processes macros. For all macros currently handled, *snacc* converts type definitions inside the macro to type references and puts the type definition in the normal scope. This way, the code generator does not have to know about macros to generate code for the types defined within them.

The only macro that receives any special processing is the SNMP OBJECT-TYPE macro. This macro's information defines an OBJECT IDENTIFIER or INTEGER to type mapping for use with any ANY DEFINED BY type. Note that the OBJECT-TYPE macro has been extended beyond its SNMP definition to allow integer values for INTEGER to type mappings.

ASN.1 allows you to define new macros within an ASN.1 module; this can change the grammar of the ASN.1 language. Since *snacc* is implemented with *yacc* and *yacc* grammars cannot be modified easily during runtime, *snacc* cannot change its parser in response to macro definitions it parses.

Any macro that *snacc* can parse has been explicitly added to the *yacc* grammar before compiling *snacc* . When a macro that *snacc* can parse is parsed, a data structure that holds the relevant information from the macro is added to the parse tree. The type and value linking passes as well as the macro processing and possibly the normalization pass need to be modified to handle any new macros that you add.

The following macros are parsed:

- OPERATION (ROS)
- ERROR (ROS)
- BIND (ROS)
- UNBIND (ROS)
- APPLICATION-SERVICE-ELEMENT (ROS)
- APPLICATION-CONTEXT
- EXTENSION (MTSAS)
- EXTENSIONS (MTSAS)
- EXTENSION-ATTRIBUTE (MTSAS)
- TOKEN (MTSAS)
- TOKEN-DATA (MTSAS)
- SECURITY-CATEGORY (MTSAS)
- OBJECT (X.407)
- PORT (X.407)
- REFINE (X.407)
- ABSTRACT-BIND (X.407)
- ABSTRACT-UNBIND (X.407)
- ABSTRACT-OPERATION (X.407)

- ABSTRACT-ERROR (X.407)
- ALGORITHM (X.509)
- ENCRYPTED (X.509)
- PROTECTED (X.509)
- SIGNATURE (X.509)
- SIGNED (X.509)
- OBJECT-TYPE (SNMP)

However, no code is generated for these macros. As stated above, only the OBJECT-TYPE macro affects the encoders and decoders.

5.8 Pass 7: Normalizing Types

The sixth pass normalizes the types to make code generation simpler. The following is done during normalization:

1. COMPONENTS OF types are replaced with the contents of the SET or SEQUENCE components that they reference.
2. SELECTION types are replaced with the type they reference.
3. SEQUENCE, SET, CHOICE, SET OF and SEQUENCE OF definitions embedded in other types are made into separate type definitions.
4. For modules in which "IMPLICIT TAGS" is specified, tagged type references such as "`[APPLICATION 2] Foo`" are marked IMPLICIT if the referenced type ("`Foo`" in this case) is not an untagged CHOICE or untagged ANY type.

5. INTEGERs with named numbers, BIT STRINGs with named bits and ENUMERATED types embedded in other types are made into separate type definitions.

The COMPONENTS OF and SELECTION type simplifications are obvious but the motivation for the others may not be so obvious. The third type of simplification makes type definitions only one level deep. This simplifies the decoding routines since *snacc* uses local variables for expected lengths, running length totals and tags instead of stacks.

The implicit references caused by "IMPLICIT TAGS" are marked directly on type references that need it. This saves the code generators from worrying about whether implicit tagging is in effect and which types can be referenced implicitly.

The types with named numbers or bits are made into a separate type to allow the C++ back end to simply make a class that inherits from the INTEGER or BIT STRING class and defines the named numbers or bits inside an `enum` in the new class.

5.9 Pass 8: Marking Recursive Types

This pass marks recursive types and checks for recursion related errors. To determine whether a type definition is recursive, each type definition is traced to its leaves, checking for references to itself. Both local and imported type references within a type are followed to reach the leaves of the type. A leaf type is a simple (non-aggregate) built-in type such as an INTEGER or BOOLEAN. At the moment, recursion information is only used during the type dependency sorting pass.

Snacc attempts to detect two types of recursion related errors. The first type of error results from a recursive type that is composed solely of type references. Types of this form contain no real type information and would result in zero-sized values. For example the following recursive types will generate this type of warning:

```
A ::= B
```

```
B ::= C

C ::= A
```

The other recursion related error results from a type whose value will always be infinite in size. This is caused by recursion with no optional component that can terminate the recursion. If the recursion includes an OPTIONAL member of a SET or SEQUENCE, a CHOICE member, or a SET OF or SEQUENCE OF, the recursion can terminate.

Both of the recursion errors generate warnings from *snacc* but will not stop code generation.

5.10 Pass 9: Semantic Error Checking

The ninth pass checks for semantic errors in the ASN.1 specification that have not been checked already. Both the type linking pass and the recursive type marking pass do some error checking as well. *Snacc* attempts to detect the following errors in this pass:

- Elements of CHOICE and SET types must have distinct tags.
- CHOICE, ANY and ANY DEFINED BY types cannot be implicitly tagged.
- Type and value names within the same scope must be unique.
- Field names in a SET, SEQUENCE or CHOICE must be distinct. If a CHOICE is a member of a SET, SEQUENCE or CHOICE and has no field name, then the embedded CHOICE's field names must be distinct from its parents to avoid ambiguity in value notation.
- An APPLICATION tag code can only be used once per module.
- Each value in a named bit list (BIT STRINGs) or named number list (INTEGERs and ENUMERATED) must be unique within its list.

- Each identifier in a named bit list or named number list must be unique within its list.
- The tags on a series of one or more consecutive OPTIONAL or DEFAULT SEQUENCE elements and the following element must be distinct.
- A warning is given if an ANY DEFINED BY type appears in a SEQUENCE before its identifier or in a SET. These would allow encodings where the ANY DEFINED BY value was prior to its identifier in the encoded value; ANY DEFINED BY values are difficult to decode without knowing their identifier.

Snacc does not attempt to detect the following errors due to the limitations of the value parser.

- SET and SEQUENCE values can be empty ({}) only if the SET or SEQUENCE type was defined as empty or all of its elements are marked as OPTIONAL or DEFAULT.
- Each identifier in a BIT STRING value must be from that BIT STRING's named bit list (this could be done in an improved value linker instead of this pass).

5.11 Pass 10: Generating C/C++ Type Information

This pass fills in the target language type information. The process is different for the C and C++ back ends since the C++ ASN.1 model is different and was developed later (more design flaws have been corrected for the C++ backend).

For C and C++ there is an array that contains the type definition information for each built-in type. For each built-in ASN.1 type, the C array holds:

typename the C *typedef* name for this type definition.

isPdu TRUE if this type definition is a PDU. This is set for types used in ANY and ANY DEFINED BY types and those indicated by the user via compiler directives. Additional interfaces to the encode and decode routines are generated for PDU types. The SNMP OBJECT-TYPE macro is the current means of indicating whether a type is used within an ANY or ANY DEFINED BY type.

isPtrForTypeDef TRUE if other types defined solely by this type definition are defined as a pointer to this type.

isPtrForTypeRef TRUE if type references to this type definition from a SET or SEQUENCE are by pointer.

isPtrForOpt TRUE if OPTIONAL type references to this type definition from a SET or SEQUENCE are by pointer.

isPtrInChoice TRUE if type references to this type definition from a CHOICE are by pointer.

optTestRoutineName name of the routine to test whether an OPTIONAL element of this type in a SET or SEQUENCE is present. It is usually just the name of a C macro that tests for NULL.

printRoutineName name of this type definition's printing routine.

encodeRoutineName name of this type definition's encoding routine.

decodeRoutineName name of this type definition's decoding routine.

freeRoutineName name of this type definition's freeing routine.

The C++ type definition array is similar to C's. It contains:

classname holds the C++ *class* name for this type definition.

isPdu same as C isPdu except that is does not affect the code generation since the C++ back end includes the extra PDU encode and decode routines by default.

isPtrForTypeDef same as C isPtrForTypeDef.

isPtrForOpt same as C isPtrForOpt.

isPtrInChoice same as C isPtrInChoice

isPtrInSetAndSeq whether type references to this class from a SET or SEQUENCE are by pointer.

isPtrInList whether type references to this class from a SET OF or SEQUENCE OF are by pointer.

optTestRoutineName name of the routine to test whether an OPTIONAL element of this type in a SET or SEQUENCE is present. It is usually just the name of a C macro that tests for NULL.

The first step of this pass uses the type arrays to fill in the C or C++ type definition information for each module's ASN.1 type definitions. This is done for the useful types module as well.

The next step goes through each constructed type and fills in the type reference information for each reference to a built-in, user defined or useful type. Much of the type reference information is taken from the referenced type's definition information. The type reference information contains the following (for both C and C++):

fieldName field name for this type if it is referenced from a CHOICE, SET or SEQUENCE.

typeName type name of the referenced type.

isPtr whether this reference is by pointer.

namedElmts named elements for INTEGER, ENUMERATED or BIT STRING types with their C names and values.

choiceIdValue if this type reference is in a CHOICE, this holds the value of the CHOICE's choiceId that indicates the presence of this field.

choiceIdSymbol if this type reference is in a CHOICE, this holds the C enum value symbol that has the choiceIdValue value.

optTestRoutineName name of the routine or macro to test for the presence of this element if it is an OPTIONAL element of a SET or SEQUENCE.

5.12 Pass 11: Sorting Types

This pass sorts the type definitions within each module in order of dependence. ASN.1 does not require the types to be defined before they are referenced, but both C and C++ do. Without this pass, the generated types/classes would probably not compile due to type dependency problems. There is no attempt to order the modules; command line order is used for the module dependence. If there are problems with mutually dependent modules, the simplest approach is to combine the dependent modules into a single ASN.1 module.

Some compilers such as CASN1 [Neufeld90] require the user to order the types within the ASN.1 modules. This can be tedious and since *snacc* may generate new type definitions from nested aggregate type definitions in the normalization pass, the user does not have complete control over the order of every type definition.

Snacc attempts to sort the types from least dependent to most dependent using the following ad-hoc algorithm:
First, separate the type definitions within a module into the following groups:

1. Type definitions that are defined directly from simple built-in types such as INTEGER.

2. Types such as SET, SEQUENCE, SET OF, SEQUENCE OF and CHOICE that contain no references to types defined in this module. That, is they are defined from only simple built-in types, imported types or useful types.

3. Type definitions that reference locally defined types.

4. Type definitions that are not referenced by any local types.

Only the 3rd group of type definitions needs more sorting. After it has been sorted, the groups are merged in the order 1, 2, 3, 4 to yield a sorted type definition list.
Now we describe how the 3rd group of type definitions is sorted.

1. For each type definition in the third group, a list of its local type references is built and attached to it. This type reference list only goes one level deep; it does not follow type references to find more type references.

2. All of the linearly-dependent types are removed and sorted. This is done by repeatedly removing type definitions that do not directly depend on any other type definitions that remain in the 3rd group. The process of removing the type definitions sorts them.

3. The type definitions that were not removed in step 2 are divided into two groups: recursive and non-recursive. The non-recursive types depend on the recursive ones since they are still in the list after step 2.

4. The non-recursive types from step 3 are sorted as in step 2. All of them should sort linearly since none are recursive.

5. If the target language is C, any SET OF or SEQUENCE OF types are separated from the recursive type definitions built in step 3. This is done because the C representation of a list type is generic (uses a *void** to reference the list element) and therefore does not really depend on the list's element type.

6. The list of local type references for the recursive types from step 3 is re-generated as in step 1 using a relaxation: types referenced as pointers are not added to a type's reference list.

7. The recursive types from step two are re-sorted as in step 2 using their new local type reference lists. Two lists are formed, those that sorted linearly and those that did not. The latter list should be empty.

To form a sorted third group, the lists are merged in the following order:

- linearly sorted types from step 2
- separated list types (C only) from step 5
- sorted recursive types from step 7
- unsorted recursive types from step 7 (hopefully empty)
- sorted non-recursive types from step 4

In C, the code generator defines both `typedef` names and `struct` tags (names). For example,

```
Foo ::= SET { a INTEGER, b BOOLEAN }

Bar ::= SEQUENCE { a OBJECT IDENTIFIER, b Foo }
```

translates to the following C data types:

```
typedef struct Foo /* SET */
{
    AsnInt a; /* INTEGER */
    AsnBool b; /* BOOLEAN */
} Foo;

typedef struct Bar /* SEQUENCE */
{
    AsnOid a; /* OBJECT IDENTIFIER */
    struct Foo* b; /* Foo */
} Bar;
```

Note that both the `struct` and the `typedef` have the name `Foo`. Also note that the `Bar` type references the `Foo` via `struct Foo*`.

For types such as `Bar` that contain the `Foo` type, `Foo` is referenced as `struct Foo*` instead of just `Foo*` because C allows you to use the type `struct Foo*` (incomplete type) in defining types even prior to the actual declaration of the the `struct Foo`. The `Foo` type can only be used after the `Foo typedef` declaration. The use of incomplete types can often overcome recursion related type ordering problems (not relevant in this example since they are not recursive).

5.13 Pass 12: Generating Code

This pass creates and fills the source files with C or C++ code. The purpose of the normalization, sorting and error detection passes is to simplify this pass. The normalization pass simplified the ASN.1 types in various ways to make C/C++ type and code generation simpler. The type sorting pass hopefully eliminates type dependency problems in the generated code. The C/C++ type generator simply proceeds through the ordered type list writing the C/C++ type definitions to a header file.

The error detection and linking passes will make *snacc* exit if errors are found, so the code generation pass can assume the ASN.1 types are virtually error free. This usually allows *snacc* to exit gracefully instead of crashing due to an undetected error.

Chapter 6

Results and Evaluation

We did a number of experiments to evaluate the performance of different encoding rules and different implementations. We also ran experiments to evaluate the performance cost of various buffer and memory management features.

The encoding rules we examined are HP/Apollo's NDR, Sun Microsystem's XDR, BER and PER (BASIC-ALIGNED). We also tested the simple and limited C based encoding rules to provide information on the maximum throughput that can be achieved if many assumptions are made. These encoding rules were described in Chapter 2.

We tested the implementations provided by five tools. The *NIDL* compiler produces C based NDR encoders and decoders. *Rpcgen* produces C based XDR encoding and decoding routines. The other three tools, *snacc*, *CASN1* and *ISODE*'s *POSY/PEPY*, compile ASN.1 to produce BER routines in C. Hand-coded implementations were used for PER and C based encoding rules

The encoders and decoders were compared for their throughput, the size of the encoded values and the size of their object code. The throughput is the most important aspect of these implementations. The size of encoded values can be important when dealing with expensive networks or for archiving purposes. The object code size of an implementation becomes a consideration when building embedded applications such as SNMP ASN.1 management for a network device.

This chapter consists of four parts:

- The comparison and analysis of different encoding rules and implementations with

a structured value.

- The comparison and analysis of different encoding rules with the integer, real and string primitive types.
- The evaluation of the performance cost of various memory and buffer management techniques.
- The execution analysis (profiling) of *snacc* BER encoders and decoders.

6.1 The PersonnelRecord Benchmark

6.1.1 Tools Benchmarked

The BER encoders and decoders of three ASN.1 tools, *snacc*, *CASN1* and *ISODE*'s *POSY/PEPY*, were benchmarked and analyzed. We also benchmarked the encoders and decoders generated by *rpcgen* (XDR) and *NIDL* (NDR). These tests are shown in Table 6.5.

For each of these tools except *snacc* , we used the "out of the box" implementation they produced. For configuration options that had to be set by the user, such as the buffering mechanism, the fastest option was chosen. Section 6.2 looks into the costs of implementation features such as these.

Three very different BER implementations from the *snacc* tool were used. We tested the C, C++ and table based versions that *snacc* produced.

NIDL and *rpcgen* were described in Chapter 2 and *snacc* was discussed in depth in Chapters 4 and 5. The *CASN1* and *ISODE* tools are briefly described here.

CASN1 [Neufeld90] is an older ASN.1 compiler developed at UBC. It produces C encoders and decoders from an ASN.1 specification. It has been used in at least one

tool	language	encoding rules
rpcgen	XDR	XDR
NIDL	NIDL	NDR
ISODE	ASN.1	BER
CASN1	ASN.1	BER
snacc	ASN.1	BER
(by hand)	ASN.1	PER
(by hand)	ASN.1	old PER
(by hand)	C	ptr to offset

Table 6.5: Tools and encoding rules tested with the PersonnelRecord Benchmark

commercial application, and in several other research projects. *CASN1* supports 1984 ASN.1 and some ROSE [ROS88] macros.

ISODE [Rose90] is a freely available ISO protocol development environment. Because of the development-oriented nature of *ISODE* , flexibility was more of a design consideration than performance. *ISODE* includes several ASN.1 tools. The *POSY* tool accepts an ASN.1 abstract syntax and produces C data structures for the ASN.1 types and an input file for *PEPY*. The *PEPY* input file consists of the original abstract syntax augmented with compiler directives, action statements, control statements and value passing statements that tell *PEPY* how to create correct BER encoding and decoding routines for the C data structure produced by *POSY*.

PEPY supports type references to other ASN.1 modules provided that the referenced ASN.1 module has already been processed by *PEPY*. It produces a file containing tagging and other information for each ASN.1 file it processes. Macro support requires the use of other tools such as *ROSY* that remove the macros from the abstract syntax and replace them with the appropriate type definitions as necessary so the resulting file can be processed by *POSY/PEPY*.

6.1.2 Benchmark Design

The aim of this benchmark is to compare the performance of different data representations and implementations thereof in handling a structured data value. We chose a "PersonnelRecord" type shown in Figure 6.7, with the value shown in Figure 6.8 for this benchmark. In retrospect, a better choice of data structure may have been the *Mix-Tree* type from [Huitema92] that allows arbitrarily "deep" values. Perhaps an even better choice would be an X.400 PDU, but specifying it in anything other than ASN.1 (e.g. *NIDL*) would be a difficult task.

To highlight the costs of the structuring mechanisms and the overall implementation, the PersonnelRecord is mostly composed of string types. The simple string types have similar translation costs for all of the encoding rules tested because the characters in the string require no real translation. Primitive types such as integers and reals are likely to vary considerably in performance between BER, PER, XDR and NDR; these are benchmarked in the next section.

Each encoder was benchmarked on the time taken to encode the PersonnelRecord value in Figure 6.8 and free the resulting encoded value one million times. Each decoder was benchmarked on the time required to decode the encoded PersonnelRecord, delete the resulting data structure and reset the reference to the encoded data one million times.

To provide more uniform results, the size of the buffer for the encoded values was set large enough to hold the entire encoded PDU. Thus, no "buffer faults" occurred during encoding (writing) or decoding (reading).

The timings were obtained by calling the UNIX *getrusage* system call [GRU90] immediately before and after both the encode and decode loop. The timings in the tables represent the difference in user time between the first and last *getrusage* call.

All encoders and decoders, including their support libraries, were compiled with the

```
PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET
{
                        Name,
    title           [0] IA5String,
                        EmployeeNumber,
    dateOfHire      [1] Date,
    nameOfSpouse    [2] Name,
    children        [3] IMPLICIT SEQUENCE
                        OF ChildInformation
                        DEFAULT {}
}

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

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE
{
    givenName  IA5String,
    initial    IA5String,
    familyName IA5String
}

EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER

Date ::= [APPLICATION 3] IMPLICIT IA5String  -- YYYYMMDD
```

Figure 6.7: ASN.1 definition of the PersonnelRecord

```
{
    {
        givenName  "John",
        initial    "E",
        familyName "Smith"
    },
    title "The Big Cheese",
    99999,
    dateOfHire "19820104",
    nameOfSpouse
    {
        givenName  "Mary",
        initial    "L",
        familyName "Smith"
    },
    children
    {
        {
            {
                givenName  "James",
                initial    "R",
                familyName "Smith"
            },
            dateOfBirth "19570210"
        },
        {
            {
                givenName  "Lisa",
                initial    "M",
                familyName "Smith"
            },
            dateOfBirth "19590621"
        }
    }
}
```

Figure 6.8: PersonnelRecord Value used in the benchmarks

MIPS cc compiler, using the `-O` compiler switch. All benchmarks were run on a MIPS R3260 (18.2 specmarks) with 48 megabytes of RAM, running version 4.51 of the MIPS UNIX operating system. The tests were run during a time when the system usage was minimal, however, this did not appear to affect the results.

Table 6.6 holds a brief description of each benchmark we performed for this section. Each benchmark (referenced by its "benchmark id" from Table 6.6) is defined concisely in the next sections. All BER encoders used the definite length option.

We tested the optimal case for XDR and NDR. The optimal case for XDR is encoding and decoding on a machine with four byte words, big endian integers and IEEE float and double representation. NDR is always optimal for encoding because it uses the host's representation. Our NDR decoding benchmark was optimal because the decoding host was the same architecture (actually the same machine). The next section looks at the non-optimal case for NDR and XDR.

The *snacc* C encoders and decoders for BER, PER91 and PER92 in this test were minimal implementations. The buffers were simply a contiguous block of memory and the nibble memory system had its expansion capability disabled. The buffer reading and writing routines did no range checking, allowing buffer overflows. Section 6.3 examines the costs of using safer buffer and memory systems. The *snacc* C++ and table based versions used in this benchmark were considerably more robust.

6.1.3 Benchmark Results and Analysis

The benchmark results for encoding and decoding the PersonnelRecord value shown in Figure 6.8 one million times are shown in Table 6.8. The memory usage during the benchmark for each implementation is shown in Table 6.9. The code sizes are shown in Table 6.7. The results are briefly presented here.

The C based encoding rules were by far the fastest, but the encoded size was the

benchmark id	encoding rules	description
1.01	BER	*POSY/PEPY* encoder and decoder with no modifications. String type "Presentation Stream" (buffer). *ISODE* version 6.
1.02	BER	*CASN1* encoder and decoder with no modifications.
1.03	XDR	XDR encoder and decoder with no modifications.
1.04	NDR	*NIDL* generated NDR encoder and decoder. Version 1.51 of the *NIDL* compiler.
1.05	C	hand coded C pointer to offset encoder and decoder.
1.06	BER	minimal *snacc* encoder and decoder.
1.07	PER91	minimal *snacc* encoder and decoder hand modified to support 1991 Packed Encoding Rules (obsolete).
1.08	PER92	minimal *snacc* encoder and decoder hand modified to support 1992 Packed Encoding Rules.
1.09	BER	*snacc* C++ encoder and decoder.
1.10	BER	minimal *snacc* table driven encoder and decoder in C.

Table 6.6: Benchmark descriptions

benchmark id	enc/dec .o	executable
1.01	12044	49152
1.02	3876	61440
1.03	884	40960
1.04	4028	159744
1.05	660	32768
1.06	8020	45056
1.09	34708	212992
1.10	0†	86016

†Table based tools have no abstract sytax specific code. A 660 byte type table was loaded for the PersonnelRecord instead.

Table 6.7: Code size for the stripped .o files of PersonnelRecord specific code and the stripped benchmark executable

benchmark id	encoding time (s)	decoding time (s)	encoded size (bytes)
1.01	982.51	1291.71	143
1.02	826.46	403.19	143
1.03	186.54	311.67	180
1.04	165.65	78.74	145
1.05	7.62	8.19	248
1.06	114.70	230.64	143
1.07	91.17	176.93	119
1.08	89.77	122.92	101
1.09	80.73	330.20	143
1.10	473.41	894.79	143

Table 6.8: Timings for encoding and decoding the PersonnelRecord 1 million times

benchmark id	local	intermediate	transfer
1.01	42/560	47/1552	1/143
1.02	32/436	152/1260†	1/143
1.03	17/208	0	1/180
1.04	1/2056	0	1/145
1.05	1/248	0	1/248‡
1.06	26/328	0	1/143
1.07	26/328	0	1/119
1.08	26/328	0	1/101
1.09	-	-	1/143
1.10	43/396	0	1/143

†An intermediate representation was only used during encoding in CASN1.

‡The transfer value is in the same memory as the local value.

Table 6.9: Memory usage in # pieces memory/total # bytes for the local, intermediate and transfer (encoded) data representation

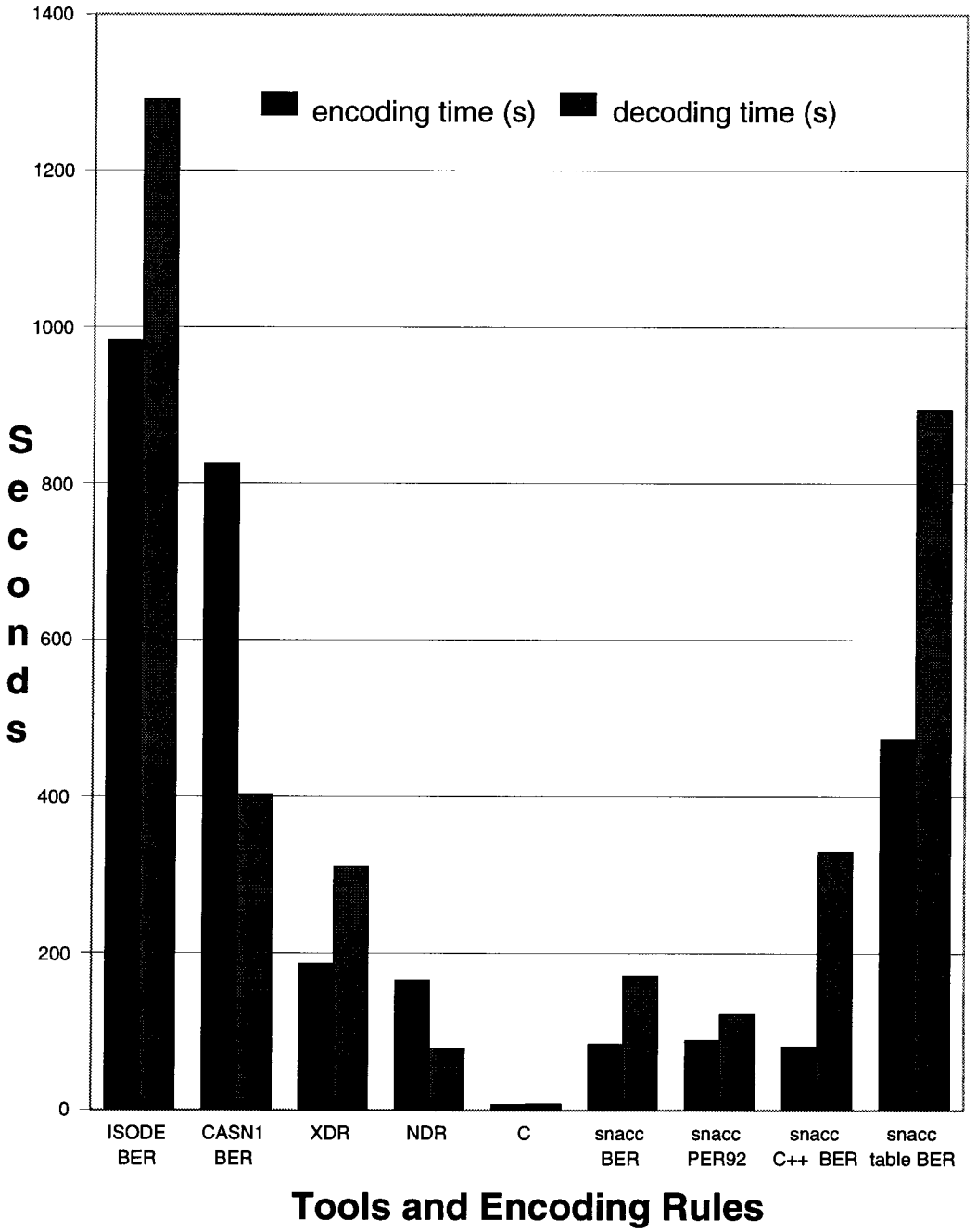


Figure 6.9: Time to encode and decode a PersonnelRecord 1 million times

largest, over twice as large as PER's, and the limitations of C make them non-viable for protocol implementation. The C encoding rules managed over 240 Mbps for encoding and decoding.

PER92, the fastest standard encoding, only managed approximately 12 Mbps (projected speed if macros are used for buffer I/O and memory allocation). PER92 also produced the smallest encoding. The PER92 decoder was third, after CER and NDR. The NDR decoder's speed can be attributed to two things. First, the NDR encoding was already in the decoding host's native format so no real data conversion was required. Second, there were no allocations required for the local data value since the limitations of NDR permit the use of a flat local data structure (i.e. the size of the local data structure is fixed so it can easily be pre-allocated).

The poor performance of the *PEPY/POSY* encoder and decoder and the *CASN1* encoder can be directly attributed to their use of an intermediate data representation between the local and transfer representations (see Table 6.9).

The table based implementation was roughly 3-4 times slower than the best of the compiled BER, PER, NDR and XDR versions. Surprisingly, the *snacc* C++ encoder implementation performed better than the stripped down *snacc* C version. Unlike the C version, the C++ implementation used safe buffer management and even used "expensive" C++ features such as virtual functions. The gcc C++ compiler may use more effective optimizations than the MIPS C compiler. The C++ decoder was considerably slower than the C version; this can most likely be attributed to the use of C++ built in new and delete operators versus the light-weight nibble memory.

For the PersonnelRecord data structure PER92 is both faster and produces smaller encodings than BER. Both the PER and BER PersonnelRecord tested contained 16 octet strings and 1 INTEGER and decoders allocated the same number of elements. The difference is in the number of tags and lengths that were encoded, resulting in the speed

and size difference. BER encoded 30 tags, 13 constructed value lengths, and 17 primitive value lengths, resulting in a 143 byte encoding. PER encoded no tags, the number of elements in the SEQUENCE OF, and 17 primitive value lengths yielding a 101 byte encoding. If the sizes of types such as the Date and the initial in the Name had been given in the ASN.1 via the SIZE subtype, their lengths would not need to be encoded by PER, allowing further compression.

Using intermediate data representations (*CASN1* and *POSY/PEPY*) greatly slows encoding and decoding. This is clearly seen by comparing the three examples using intermediate data structures to the snacc version that has similar features except for the intermediate data structure. CASN1's encoder (Benchmark 1.02 vs. 3.04) was over 7 times slower, POSY/PEPY's encoder (Benchmark 1.01 vs. 3.08) was 8.5 times slower, and the POSY/PEPY decoder was over 4 times slower. Encoding rules should be designed to support one pass encoding and decoding, eliminating the need for costly intermediate data structures. One pass, forwards encoding allows an application to encode directly to a stream oriented protocol such as TCP.

Table 6.7 shows the code sizes of the stripped `.o` files for the PersonnelRecord specific files, and the sizes of the stripped benchmark programs. The most important result is that table based encoders and decoders have no code specific to an abstract sytnax; instead they load a type table. The type tables are considerably smaller that their corresponding code. The PersonnelRecord type table was only 660 bytes. The code size of the *snacc* generated encoders, decoders, print and free routines for the X.500 '88 ASN.1 specification was 450 kilobytes. The type table for the same specification was only 27 kilobytes.

POSY/PEPY

POSY/PEPY encoders consist of two separate stages: the conversion of a value in an

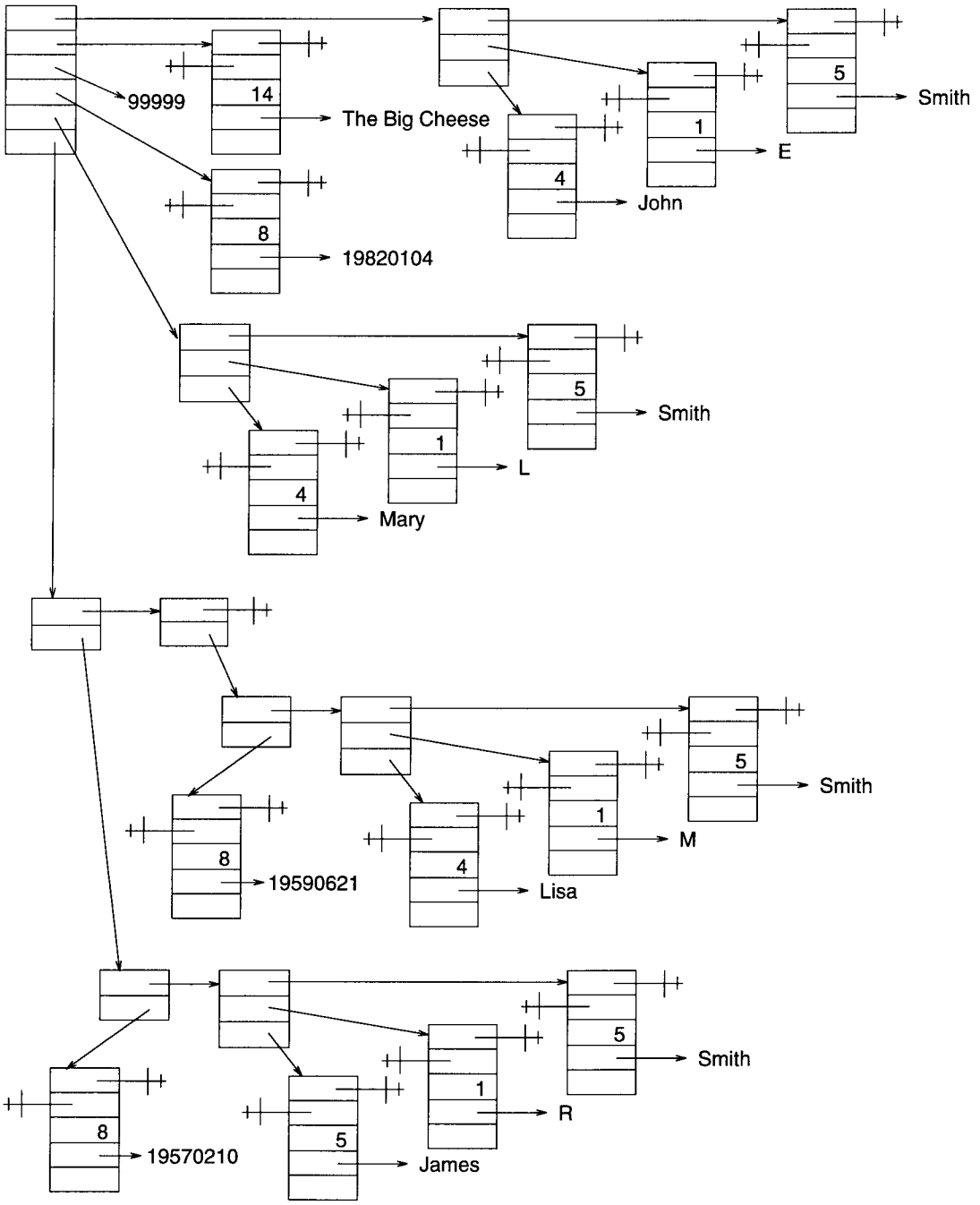


Figure 6.10: ISODE's local representation for the PersonnelRecord Value

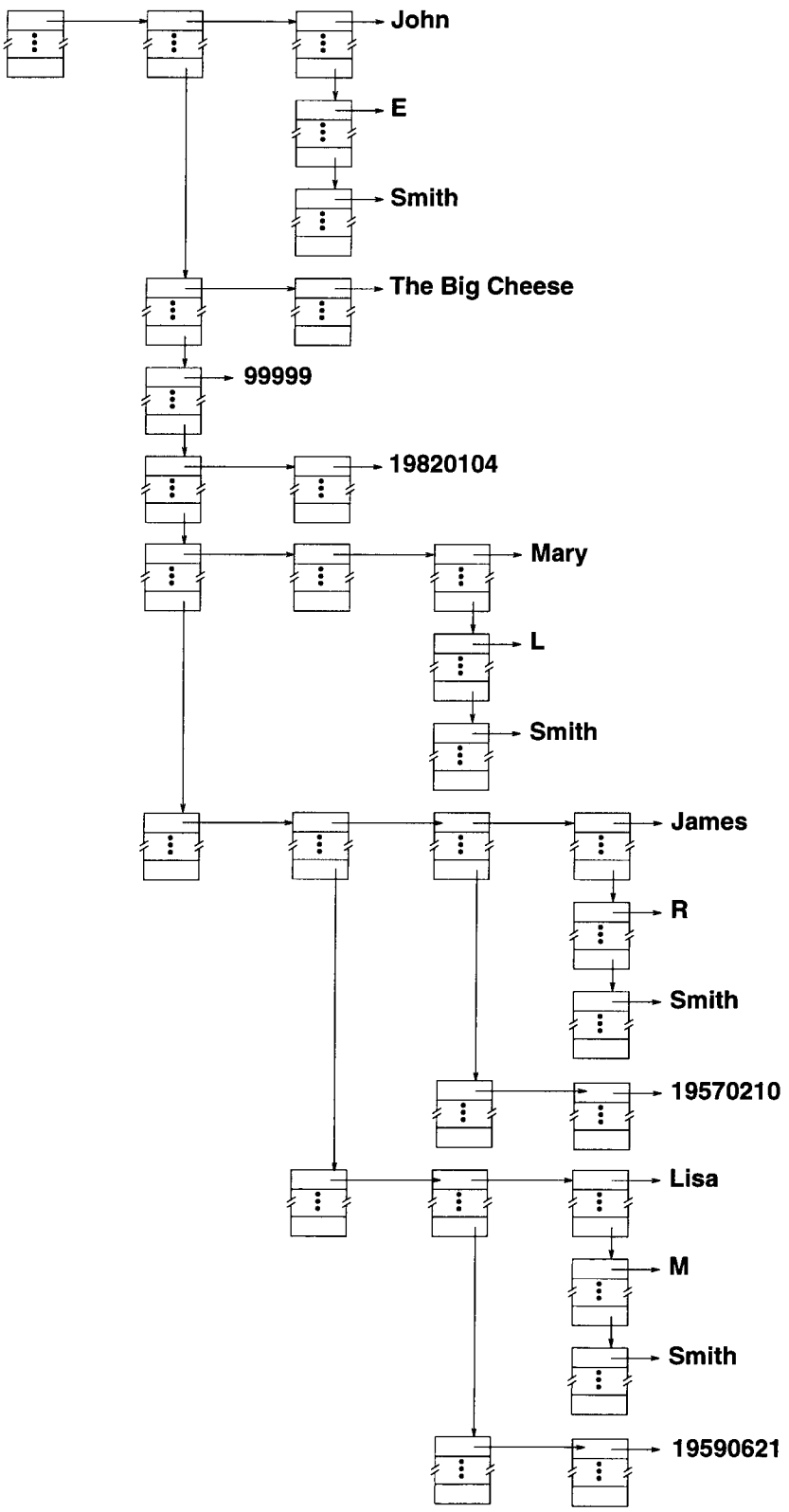


Figure 6.11: ISODE's PElement representation for the PersonnelRecord Value

abstract syntax specific data structure (see Figure 6.10) into the BER based PElement tree (see Figure 6.11) and then conversion of the PElement tree into the send buffer format (a contiguous block for benchmarking purposes). Each PElement contains a tag, a length, a pointer to its data (primitive data or a child PElement), a pointer to its sibling (next element in the same SET or SEQUENCE) and other information. The use of the PElement tree is costly; each PElement is 48 bytes (see Table 6.9). The decoders use the inverse of the encoding process.

ISODE encoders and decoders provide flexible buffer management that supports different buffer types at runtime by using the "Presentation Stream" data structure. The Presentation Stream data type contains pointers to the buffer routines and data; calling buffer functions indirectly will hurt performance. The memory primitives used for the decoded values and the PElements were malloc and free. Data items are freed by hierarchically traversing the data structure.

CASN1

CASN1's approach to encoding is to encode forwards into small, linked "IDX" buffers, maintaining a stack of buffer pointers and a corresponding stack of lengths for length calculation of constructed types (see Figure 6.13). When the components of a constructed value have been encoded, the length on the length stack is encoded into its own IDX buffer and inserted after the buffer on the buffer stack.

This encoding process results in every tag, end-of-contents and most lengths having their own IDX buffer. The ratio of memory used by the IDX structures to actual BER data is very high for some abstract syntaxes; for the 143 byte encoded length of the PersonnelRecord value used in the benchmarking, 76 IDX buffers were used. This is a 6.4 to 1 ratio between the memory used for IDX buffers and actual BER data. Even for indefinite lengths, the same length insertion process is used. The IDX buffers' data are

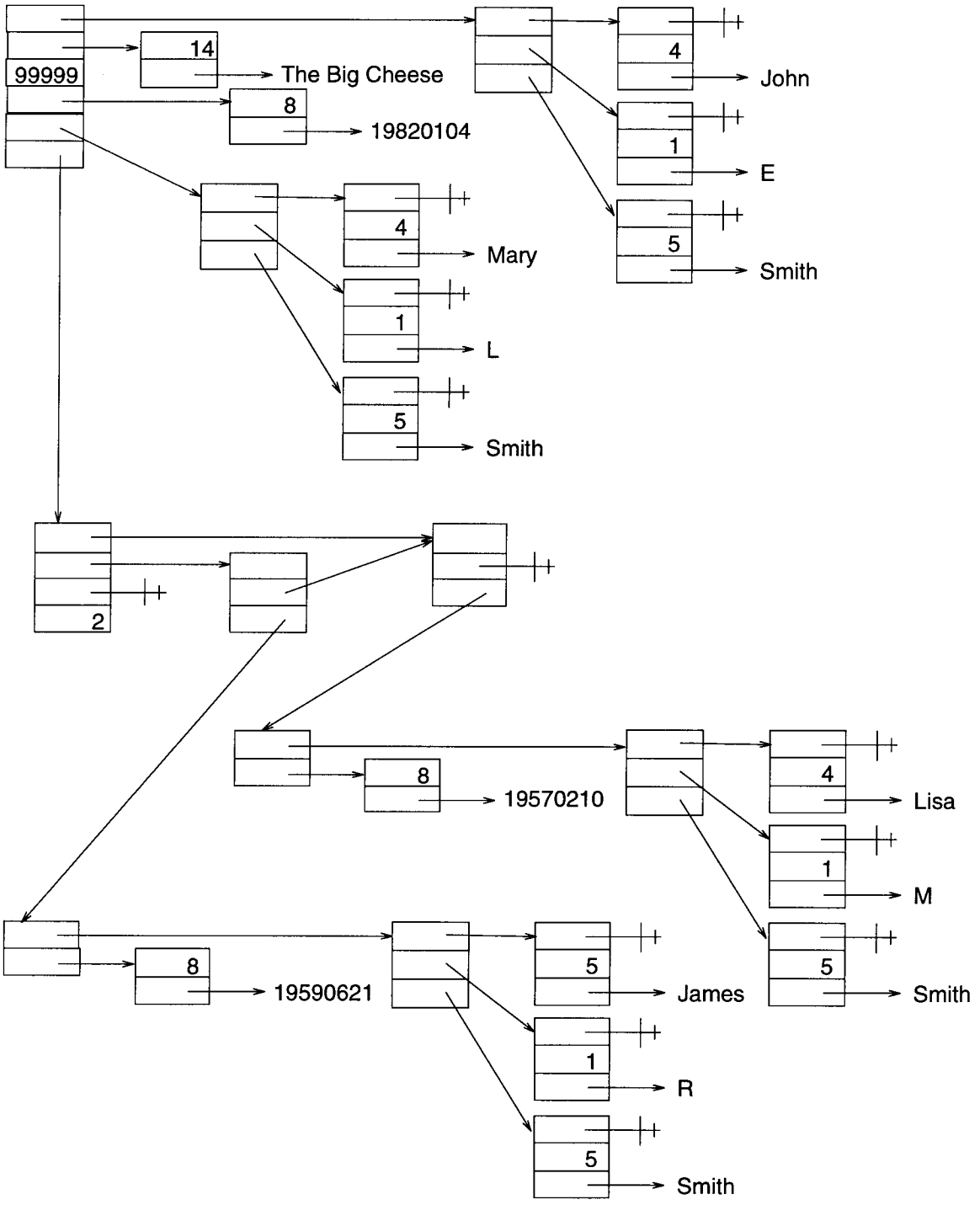


Figure 6.12: CASN1 local representation for the PersonnelRecord Value

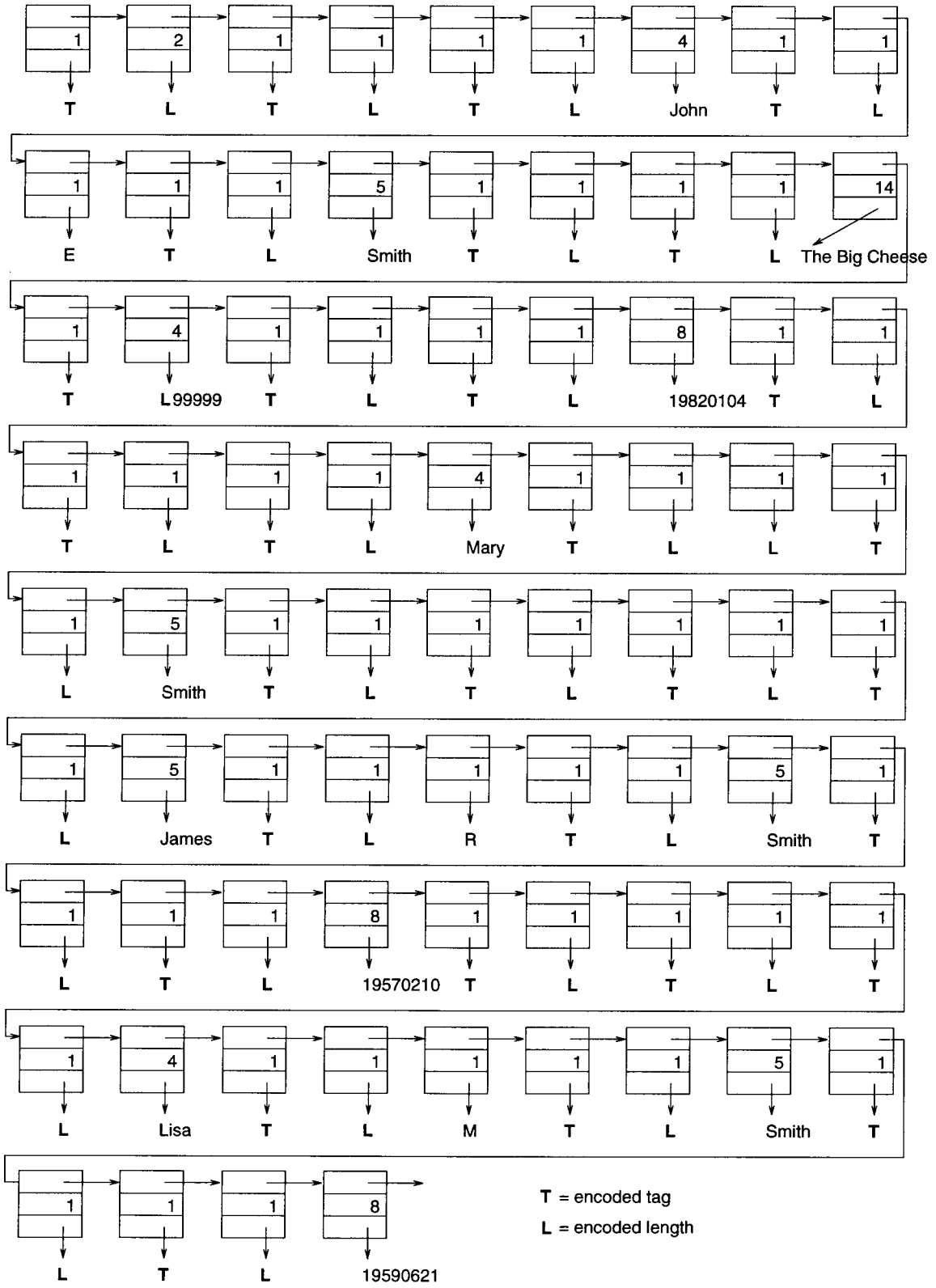


Figure 6.13: CASN1's IDX buffer structure for the PersonnelRecord Value

serialized (copied) into a single buffer for transmission.

CASN1 decoding is very similar to *snacc* decoding except tags can be decoded more than once. When decoding the elements of a SET, the SET decoder peeks ahead in the buffer and decodes the next tag to determine which element to decode. When the element decoding routine is called, the tag is decoded again. Tags may also be decoded more than once when decoding OPTIONAL elements of a SEQUENCE. When dealing with simple type definitions, such as `Foo ::= Bar`, CASN1 generates routines for the `Foo` type that simply call the `Bar` type's routines. This introduces unnecessary function calls for encoding and decoding `Foo` when they could easily be avoided by calling the `Bar` type's routines directly.

Decoders produced with CASN1 have the drawback of requiring the PDU to be decoded to be in a contiguous buffer and no range checking is done by the decoding routines. Thus, a malformed BER PDU can cause a segmentation fault (e.g. too long a length on a primitive data value).

For allocating space for decoded values, a nibble memory scheme with range checking was used when benchmarking the CASN1 decoder.

Rpcgen

The *rpcgen*/XDR version of the PersonnelRecord is shown in Figure 6.14. Note that this definition contains much more information than the ASN.1 version with respect to the local representation's format. The XDR local representation for the PersonnelRecord value is shown in Figure 6.15. Its simplicity reduces the allocation needs of the decoder.

For each type, *rpcgen* produces a single XDR routine that handles encoding, decoding and freeing. The generated routines take an "XDR" type argument that is similar to ISODE's Presentation Stream type, except that it includes the operation: encode, decode or free. The XDR routine for each type checks the "XDR" type to determine whether to

```
typedef string IA5String<>;

typedef IA5String Date;

typedef int   EmployeeNumber;

struct Name
{
    IA5String givenName;
    IA5String initial;
    IA5String familyName;
};

struct ChildInformation
{
    Name  name1;
    Date  dateOfBirth;
};

struct PersonnelRecord
{
    Name             name1;
    IA5String        title;
    EmployeeNumber   employeeNumber3;
    Date             dateOfHire;
    Name             nameOfSpouse;
    ChildInformation children<>;  /* var-sized array  */
};
```

Figure 6.14: XDR PersonnelRecord Definition

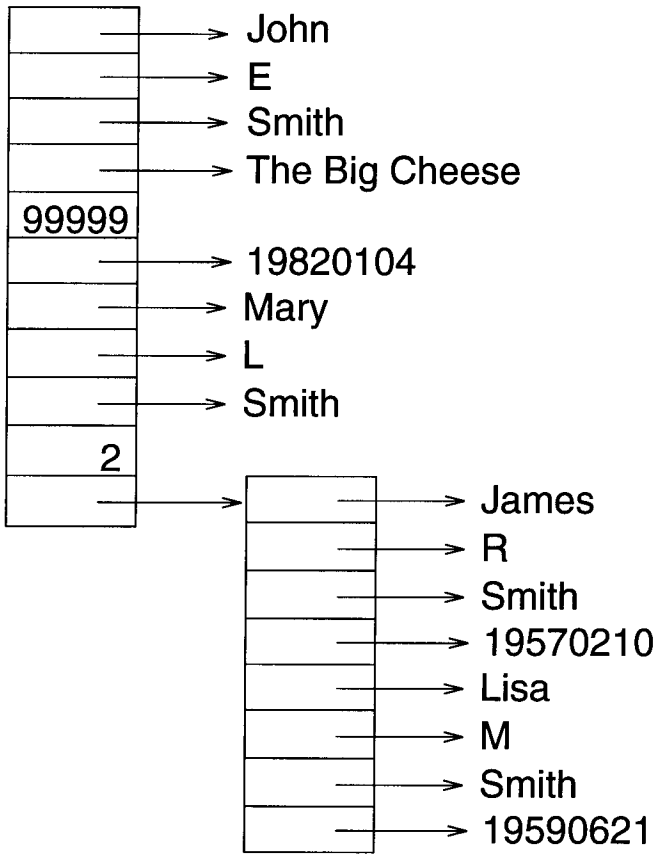


Figure 6.15: XDR's local representation for the PersonnelRecord Value

encode, decode or free.

The `xdr_mem` buffer type, a simple contiguous block, was used for the benchmarks. We benchmarked XDR with its default memory scheme where the decoded value is allocated using `malloc` and `free` using a hierarchical free. By replacing the memory management with nibble memory, the XDR decoding time improves by a factor of two.

NIDL

The *NIDL* version of the PersonnelRecord is shown in Figure 6.17. As with the XDR version, it contains much more information about the local representation's format.

The NIDL local representation for the PersonnelRecord is shown in Figure 6.16. The use of fixed-size strings makes the entire PersonnelRecord a fixed-size type. The fixed-size nature allows the NDR decoder to pre-allocate the entire PersonnelRecord; this is the main reason that the NDR decoder performed so well.

NDR encoders are a single routine that first calculates the encoded length of the given value and then encodes it into a buffer of appropriate size. NDR encoders encode the data in the local host's format and include a format identifier so the receiving host knows how to decode the data. Thus the decoding routine must choose the appropriate decoding rules on the basis of format identifier. However, for this benchmark, the format used in the PDU was the same as the decoder's internal format; this is the optimal case for NDR decoders.

Snacc

Snacc generates "backwards" BER encoders [Steedman90]. This allows *snacc* to translate directly from its local representation (see Figure 6.18) to the network format.

Indefinite lengths are encoded by writing a special octet where the definite length normally goes and writing an End-Of-Contents (EOC) marker after the content. The

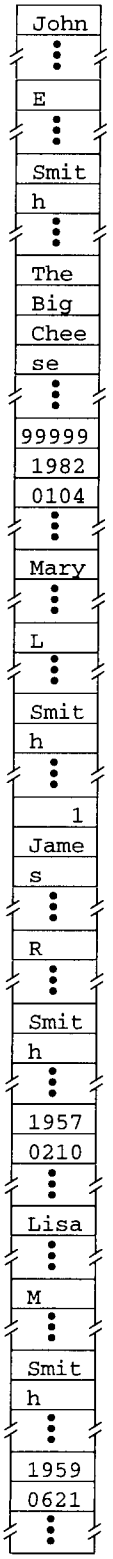


Figure 6.16: NIDL's local representation for the PersonnelRecord Value

```
typedef string0[128] IA5String; /* fixed-size locally */

typedef IA5String Date;

typedef int EmployeeNumber;

typedef struct {
    IA5String givenName;
    IA5String initial;
    IA5String familyName;
} Name;

typedef struct {
    Name name1;
    Date dateOfBirth;
} ChildInformation;

typedef [handle] struct  {
    Name name1;
    IA5String title;
    EmployeeNumber employeeNumber3;
    Date dateOfHire;
    Name nameOfSpouse;
    long lastIndex;
    ChildInformation [last_is(lastIndex)] children[];
} PersonnelRecord;
```

Figure 6.17: NIDL PersonnelRecord Definition

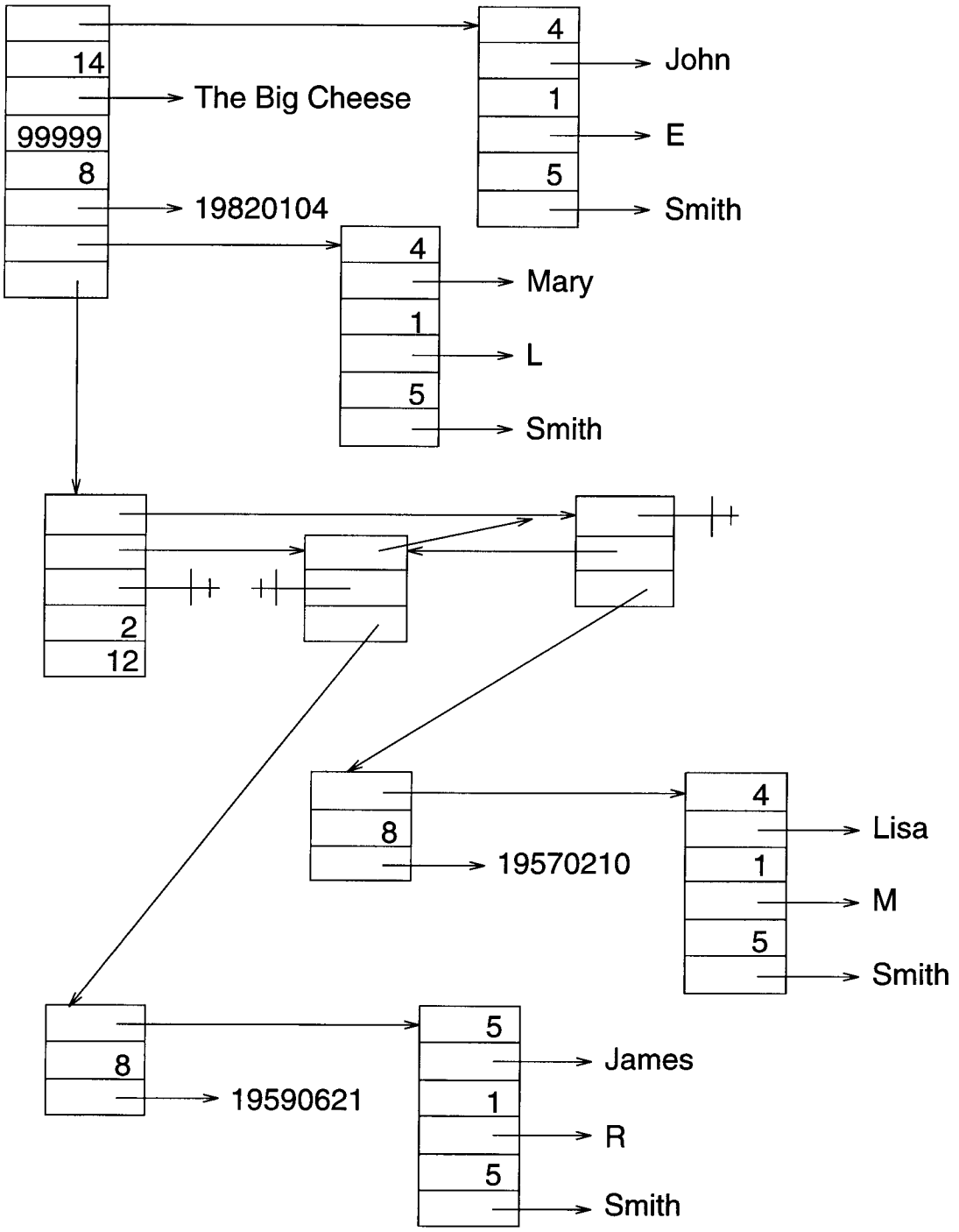


Figure 6.18: *Snacc*'s local representation for the PersonnelRecord Value

EOC is two zero octets. Indefinite lengths cannot be used on non-constructed values because the EOC may be indistinguishable from the value's data.

Note that backwards encoding requires encoding the entire PDU before sending it, therefore stream-oriented communication, where parts of the PDU are sent as they are encoded, cannot be used effectively. Using the indefinite length option for constructed values exclusively eliminates the advantage of backwards encoders at the expense of a slightly larger encoded value (see the next section). *Snacc* can use both definite and indefinite lengths with similar performance.

6.2 Primitive Type Benchmarks

The translation of the integer and real primitive types is very different in NDR, XDR, BER and PER. NDR and XDR used fixed-size representations, whereas BER and PER used variable-size representations. Simple string types do not typically involve much translation unless different character sets are used. We tested the performance of the encoding rules for integer, real and string types to confirm these assumptions. We also include the cost of a `memcpy` of the same size as the encode array for each array to provide a lower bound for performance.

This benchmark is useful for evaluating encoding rules with respect to the types used in an abstract syntax. For example, implementors of application protocols such as X.400 and X.500, whose PDUs are mostly structures and lists of strings, need not worry about the integer or real type performance. However, writers of an application, such as a distributed spread-sheet, should be very concerned about the integer and real encoding and decoding performance.

6.2.1 Benchmark Design

Table 6.10 shows the benchmarks we performed on primitive types. In this test we wanted to avoid implementation issues, so each test used the same memory and buffer management. Only the actual code that performed the encoding and decoding of the primitive type varied. This technique allows a more accurate gauge of the encoding rules themselves instead of just their implementation. In some cases (as indicated in Table 6.10) the primitive type encoding and decoding routines were functions while in other cases they were placed "inline".

No allocations or frees were done during the integer and real type benchmarks. `Malloc` and `free` were used during the string decoding benchmarks.

For each set of encoding rules, the same buffer was overwritten for each encoding iteration. This buffer was then used for each decoding iteration. The same, randomly-generated integers and reals were used for all of the integer and real type benchmarks.

As mentioned in the last section, there are certain conditions under which NDR and XDR operate optimally. In this section we look at both the optimal and non-optimal cases for the real and integer types. Since the non-optimal case of XDR affects both its encoding and decoding, we used an extra XDR benchmark to represent both the XDR and NDR non-optimal case. NDR is always optimal for encoding.

For the integer tests, we encoded and decoded one hundred random integers 10,000 times. All of the integer tests used the same local representation for the integers, an array of 100 `long int`s.

All real values were internally represented by the `double` type. The real test was conducted in the same manner with 100 random double precision real values.

All strings were represented by a `struct` that contained a `char*` for the string and a `long int` for the length of the string. The string benchmarks encoded and decoded an

array containing 100 "Hello World" strings, 10,000 times. The content of the string did not affect the encoding or decoding process in any of the cases. Each string benchmark was run as a separate program so the memory manager would be in the same state when the benchmarks started.

We also tested a fixed-size string type for PER (Benchmark 2.16 in Table 6.10). The performance of the other encoding rules for this will be the same as for the unconstrained OCTET STRING, however, PER can eliminate the length encoding for the string since it is a fixed-size. The benchmark for this type uses the same array value and number of iterations as the other string benchmarks.

For each array that was encoded, we benchmarked the cost of doing a `memcpy` of the same size as the encoded array, 10,000 times. This provides a lower bound for both encoding and decoding cost. Conceivably, encoding rules such as C based encoding rules could be used to eliminate the need for any copying (i.e. the local representation is not copied but modified in place).

BER primitive values consist of the type's (UNIVERSAL) tag, length and content. PER primitive type encodings are just the length and content except in special fixed-size cases where they consist solely of the content. The XDR and NDR integer and real types were four and eight bytes long, respectively. The XDR string encodings are preceded by a four byte length determinant. NDR strings are encoded similarly except the length determinant is only two bytes. XDR requires all components to be aligned to four byte boundaries. NDR allows 2, 4 or 8 byte alignment depending on the type.

The indefinite length form was used when encoding the BER SEQUENCE OF values.

6.2.2 Benchmark Results

For the integer array, the optimal XDR and NDR cases are almost five times faster than the PER and BER versions. However, if XDR is used on a little endian host, the

benchmark id	encoding rules	description
2.01	NDR	integer, optimal case, inline `int intArray[100]`
2.02	XDR	integer, optimal case, inline `int intArray[100]`
2.03	XDR	integer, big to little endian conversion, inline `int intArray[100]`
2.04	BER	integer, inline `SEQUENCE (SIZE 100) OF INTEGER`
2.05	BER	integer, fcn call `SEQUENCE (SIZE 100) OF INTEGER`
2.06	PER92	integer, inline `SEQUENCE (SIZE 100) OF INTEGER`
2.07	NDR	real, optimal case, inline `double realArray[100]`
2.08	XDR	real, optimal case, inline `double realArray[100]`
2.09	XDR	real, translating as if on a Vax, fcn call `double realArray[100]`
2.10	BER	real, fcn call `SEQUENCE (SIZE 100) OF REAL`
2.11	PER92	real, fcn call `SEQUENCE (SIZE 100) OF REAL`
2.12	NDR	string, inline `string0 strArray[100]`
2.13	XDR	string, inline `string strArray[100]`
2.14	BER	string, inline `SEQUENCE (SIZE 100) OF OCTET STRING`
2.15	PER92	string, inline `SEQUENCE (SIZE 100) OF OCTET STRING`
2.16	PER92	fixed size string, inline `SEQUENCE (SIZE 100) OF OCTET STRING (SIZE 11)`

Table 6.10: Primitive type benchmark definitions

benchmark id	encoding time (s)	decoding time (s)	copy time (s)	encoded size (bytes)
2.01	0.62	0.61	0.14	400
2.02	0.62	0.61	0.14	400
2.03	1.30	1.32	0.14	400
2.04	2.86	2.99	0.22	603
2.05	3.09	2.71	0.22	603
2.06	2.46	2.74	0.18	499

Table 6.11: Results for integer

benchmark id	encoding time (s)	decoding time (s)	copy time (s)	encoded size (bytes)
2.07	1.12	0.84	0.27	800
2.08	1.12	0.84	0.27	800
2.09	6.24	4.27	0.26	800
2.10	9.36	34.30	0.41	1204
2.11	8.58	34.11	0.37	1100

Table 6.12: Results for real

benchmark id	encoding time (s)	decoding time (s)	copy time (s)	encoded size (bytes)
2.12	4.28	6.05	0.52	1400
2.13	3.89	7.20	0.52	1600
2.14	3.53	6.95	0.44	1304
2.15	2.92	6.59	0.40	1200
2.16	2.54	5.81	0.37	1100

Table 6.13: Results for string

costs double due to the conversion between big and little endian format. This implies that NDR decoding time will double when the receiver's architecture has the opposite "endian-ness" from the sender.

Due to the elimination of tags, the PER integer encoding is one byte smaller for every integer encoded than the BER encoding. PER also eliminates the tag and length on the fixed-size SEQUENCE OF type. The BER and PER integer encodings are larger than the XDR and NDR versions. This is mostly due to the random number generator choosing the more common four byte values. For applications that use integers closer to zero, the BER and PER encodings may be smaller than the NDR or XDR ones.

The function call overhead difference between Benchmarks 2.04 and 2.05 did not appear to have much affect. In the encoding case, the function call overhead slowed it somewhat but in the decoding case it was actually faster. The integer and real encoders and decoders for the optimal NDR and XDR cases were very small (2 lines of C) and used no temporary variables; this simplified inlining them. The other encoding and decoding routines were more complex and larger, especially in the case of the BER and PER real routines.

The BER and PER real encoders and decoders are much slower than the XDR and NDR implementations. The optimal NDR and XDR encoders and decoders simply copy the eight byte double value to encode it. Benchmark 2.09 shows that an extra cost is incurred if the local real representation is not IEEE format (Vax G_floating in this case).

BER and PER real types performed poorly in all respects compared to the XDR and NDR versions. The most expensive operation is BER and PER real decoding. Their performance is over 40 times worse than the XDR and NDR decoders. This can be traced to the use of floating point operations (`pow` from UNIX C math library). The encoders were 9 times slower than the XDR and NDR ones. The BER and PER real encoding routines are not as bad as their decoding routines in performance because they assumed

that their host used the IEEE double format. Thus, they did not need to do floating point operations to encode.

The size of the BER and PER encodings were 1.4 to 1.5 times larger than the XDR and NDR versions. BER and PER would vastly improve their performance for real numbers if they used the IEEE `double` representation for reals.

The NDR strings are smaller than XDR due to their small, but more limiting, length determinant. NDR strings include a null-terminator character unlike XDR, however it did not affect the value we tested (XDR inserted one padding byte).

The PER and BER strings are smaller than the XDR and NDR ones because the length determinant is encoded in the minimum number of bytes and no alignment is required. The PER encoding and decoding of the fixed-size OCTET STRING was faster and smaller than the others due to the elimination of the string's length determinant.

6.3 Benchmark of Implementation Features

6.3.1 Benchmark Design

In this section we look at the implementation issues and costs for memory and buffer management as well as the cost of definite versus indefinite lengths in BER. We use both the PersonnelRecord value from Section 6.1 and an X.500 ListResult value.

To observe the impact on performance, two buffer management schemes are used with *snacc* encoders and decoders. The first scheme uses one fixed-size block for reading and writing. Two variations of this were benchmarked, one that does range checking and one that does not. Range checking (checking for buffer overrun on reads and writes) is essential for preventing segmentation faults when decoding an improperly encoded value.

The second buffer scheme uses linked buffers consisting of two separate parts, a buffer index portion and a data portion. The index part has pointers to the data block and the

next index part (if any) along with some additional information. Data blocks contain only data and are allocated from a fixed-size pool of free blocks. The index parts are allocated from another pool of index-sized blocks.

Decoders can do many allocations when building the local value while decoding so three memory allocation schemes were tested. The first method "nibbles" chunks from a fixed-size block, allowing the entire decoded value to be freed by resetting a pointer. Two variants of the nibble memory allocation scheme were tested; one that checks for allocations past the end of the block and one that does not. The second method is like the first except that more blocks for nibbling can be allocated as needed (range checking is required for this feature). The third scheme uses `malloc` with "hierarchical" freeing. Hierarchical freeing means that the data structure is freed by traversing it with freeing routines specific to that data structure.

An interesting memory management scheme that was not tested is using memory protection and mapping in paged memory systems. A sequence of contiguously mapped pages are used for a nibble block, with the last page being protected. A protection fault to this page would result in the allocation and mapping of more pages onto the end of the nibble block. This scheme could be used for buffers as well, where protection faults from writes allocate more pages or return an end-of-data indicator for reads. This method has two benefits: range checking overhead can be eliminated and contiguous data is easier to process.

We also measured the performance improvement that can be achieved from using macros instead of functions calls for memory allocation and buffer reads (decoding) and writes (encoding).

6.3.2 Benchmark Results

benchmark id	encoding rules	description
3.01	BER	minimal *snacc* encoder and decoder. Reference implementation.
3.02	BER	same as 3.01 except indefinite length form used where possible.
3.03	BER	same as 3.01 except range checking is done for buffer reads and writes.
3.04	BER	same as 3.01 except range checking is done for memory allocations (decoding only).
3.05	BER	same as 3.01 except uses linked buffer scheme.
3.06	BER	same as 3.01 except nibble memory pool can grow with demand (decoding only).
3.07	BER	same as 3.01 except buffer reads/writes and memory allocations are macros.
3.08	BER	same as 3.01 except memory management uses `malloc` and hierarchical `free`.

Table 6.14: Benchmarks of BER implementation features

benchmark id	encoding time (s)	decoding time (s)	encoded size (bytes)
3.01	114.70	230.64	143
3.02	124.60	267.19	168
3.03	141.48	256.96	143
3.04	114.69	230.42	143
3.05	163.36	272.93	143
3.06	114.66	238.96	143
3.07	84.50	171.29	143
3.08	116.26	303.93	143

Table 6.15: Results for encoding and decoding the PersonnelRecord 1 million times

benchmark id	encoding time (s)	decoding time (s)	encoded size (bytes)
3.01	46.41	88.26	5074
3.02	44.17	95.48	6152
3.03	46.60	100.46	5074
3.04	47.60	90.37	5074
3.05	61.30	146.30	5074
3.06	38.26	130.86	5074
3.07	31.16	69.77	5074

Table 6.16: Results for encoding and decoding an X.500 ListResult 10,000 times

Benchmarks 3.01 to 3.08 measure the costs of adding various features to *snacc* generated encoders and decoders. Using indefinite lengths (where possible) when encoding was 9% more expensive for the PersonnelRecord yet 5% cheaper for the ListResult. Decoding was, respectively, 16% and 8% more expensive. The sizes of the indefinite length encodings were 17% larger for the PersonnelRecord and 21% larger for the ListResult.

Benchmarks 3.01 and 3.03 show that range checking on buffer writes can increase the encoding time by up to 23%. Range checking for buffer reads can add 11 to 14% to the decoding time. Benchmarks 3.01 and 3.05 show that using a linked buffer scheme (requires range checking) that can expand as necessary, can add 32 to 42% to the encoding time and 18 to 65% to the decoding time. The performance cost of the range checking will increase as the number of writes and reads increases. For example, range checking will be more of an issue when doing 100 one byte buffer writes than when doing one 100 byte buffer write. If the encoding rules, abstract syntax or protocol permit, it is advantageous to do a simple length calculation before encoding so that the actual size of the send buffer can be determined. This allows a much simpler buffering scheme such as a single buffer with no range checking on writes to be used.

Using macros for the buffer reading and writing and memory allocation caused the

data structure	benchmark id	% time encoder uses for buffer writes	% time decoder uses for buffer reads	% time decoder uses for memory mgmt.
PersonnelRecord	3.01	47	25	8
PersonnelRecord	3.08	47	19	28
X.500 ListResult	3.01	36	18	11

Table 6.17: Overall cost of buffer and memory management for *snacc*'s BER and PER encoders and decoders

most significant speed up (21-33%), without a huge increase in the compiled size. However, the compiled size of our implementations increased dramatically if the primitive type encoding and decoding routines were macros.

An anomaly appears in Benchmark 3.06 of the ListResult (see Table 6.16). The encoding time should not be different from that of Benchmark 3.01 since the only difference between them is the allocation scheme for decoding. The difference was traced to a compiler optimization that was possible in Benchmark 3.06 but not Benchmark 3.01.

Using a `malloc`-like memory allocator with a hierarchical free instead of the simpler nibble scheme caused a 32% increase in decoding time. An environment which permits decoding into a single buffer or set of buffers that can be freed in one operation will significantly improve the performance. Such a scheme is provided in the programming environment used in implementing the EAN X.500 system [Neufeld92-2]. *Snacc* provides support for this type of memory management in its runtime library.

6.4 Execution Analysis

We used "pixie" [PIX90] basic block profiling to determine the component costs of *snacc*'s encoders and decoders (see Table 6.17). The results showed that *snacc*'s BER encoder, using an extremely simple buffer format (contiguous block, no range checking), can spend almost half of their CPU cycles simply writing to the buffer. With the same

buffer format, decoders spent up to 28% of their time just reading from the buffer. Decoders using simple nibble allocation used from 8% to 10% of their time in memory management; however, using a `malloc` and hierarchical free memory system instead pushes the cost to 28%.

Buffer and memory management accounted for less of the X.500 ListResult's encoder's and decoder's processing time than those of the PersonnelRecord. This seems to indicate that as the abstract syntax for a type becomes more complex, the cost of traversing (encoding) or assembling (decoding) the local representation value increases. The pixie profiling also indicated that only about 20% to 40% of the encoding and decoding time was spent in the routines generated by the compiler for the abstract syntax. The rest was spent in the library routines for the primitive types which in turn called the buffer and memory routines.

Chapter 7

Conclusions

Our study revealed that the majority of encoding and decoding time is spent in the following areas:

- writing to the send buffer during encoding
- reading from a receive buffer when decoding
- memory allocation for the decoded value's data structure
- freeing the decoded data structure

Each of these areas is examined to determine how they can be improved. However, even after optimization, these areas still consume a significant portion of the processor time during encoding and decoding.

The results of the analysis indicate that the majority of time is being spent in primitive type encoding and decoding routines (which contain most of the calls to the buffer and memory routines) rather than the routines generated by the compilers. It is clear from these results that efficient encoders and decoders must be tuned to work well with the buffer structure and memory model being used. Even highly tuned encoding rules such as PER92 only achieved about 12 Mbps on a MIPS R3260. Either faster processors and memory or faster encoding and decoding techniques are necessary for the presentation layer to keep up with the potential rate at which data can arrive on today's high speed

networks. Our results suggest several design rules that can aid in the development of efficient encoders and decoders:

- Avoid intermediate representations such as IDX buffers between the local representation and the fully encoded representation.
- Choose the simplest buffer management system that your encoding rules and environment will allow. Using "inline" procedures or macros is a significant advantage.
- Choose a memory allocation/free model that supports cheap allocations and frees. Avoid hierarchical freeing if possible.
- Design local data structures that minimize the number of allocations by eliminating unnecessary pointers.
- Use a simple pre-encoding length calculation if the abstract syntax, protocol or encoding rules permit, to simplify buffer management for encoders.

For the PersonnelRecord data structure, PER92 is better than BER in performance and compression. However, the extent of the improvement greatly depends on the abstract syntax of the values being processed. Abstract syntaxes that use subtyping may allow PER92 to produce smaller encodings possibly at the cost of encoding and decoding speed.

One-pass forward PER92 encoders can be implemented; this allows performance and streaming to transport connections. Finally, both BER and PER encoders and decoders performed well in comparison to XDR and NDR encoders and decoders. However, the BER and PER performance will decrease as the data becomes more numerically oriented.

The *snacc* tool generates useful, efficient implementations. Obvious future development would include handling the new language features of ASN.1 '92 and supporting

different encoding rules such as PER, XDR, and possibly C based encoding rules (useful when archiving to disk). The *snacc* generated type tables need to be improved such that they contain subtyping information; subtyping information is necessary for PER encoders and decoders.

In general, the maximum throughput of encoding rules is limited by the lower bound of a simple memory copy of the PDU. In certain, limited cases, such as C based encoding rules, this can be improved upon by encoding in place.

Future research on the performance of encoding rules is necessary. Perhaps parallelism can be used to improve their throughput. C based encoding rules demonstrated that if certain assumptions can be made, a large performance increase can be had. For example, if hardware producers could use a common machine data representation standard then this type of performance improvement could be made. The translation of primitive types, which can be expensive (e.g. BER real values), would not be necessary.

Support for commonly used data structure features, such as multiple references to the same value and even cyclic references, should be examined. Perhaps ASN.1 can be extended with new language features and encoding rules to support this.

Appendix A

C Example

This appendix contains an example of the C code *snacc* will generate for an ASN.1 data structure. *Snacc* generated two files for the following PersonnelRecord ASN.1 data structure, p_rec.c and p_rec.h.

```
P-REC DEFINITIONS ::=
BEGIN


PersonnelRecord ::= --snacc isPdu:"TRUE" -- [APPLICATION 0] IMPLICIT SET
{
                        Name,
    title           [0] IA5String,
                        EmployeeNumber,
    dateOfHire      [1] Date,
    nameOfSpouse    [2] Name,
    children        [3] IMPLICIT SEQUENCE OF ChildInformation DEFAULT {}
}

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

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE
{
    givenName  IA5String,
    initial    IA5String,
    familyName IA5String
}

EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER

Date ::= [APPLICATION 3] IMPLICIT IA5String -- YYYYMMDD

END
```

Figure A.19: PersonnelRecord ASN.1 data structure

Snacc generated C file p_rec.h

```
/*
 *    p_rec.h
 *
 *    "P-REC" ASN.1 module C type definitions and prototypes
 *
 *    This .h file was by snacc on Sun Apr 11 02:17:21 1993
 *
 *    UBC snacc written compiler by Mike Sample
 *
 *    NOTE: This is a machine generated file - editing not recommended
 */


#ifndef _p_rec_h_
#define _p_rec_h_


typedef AsnInt EmployeeNumber; /* [APPLICATION 2] IMPLICIT INTEGER */
#define BEncEmployeeNumberContent BEncAsnIntContent
#define BDecEmployeeNumberContent BDecAsnIntContent
#define PrintEmployeeNumber PrintAsnInt
#define FreeEmployeeNumber FreeAsnInt


typedef struct Name /* [APPLICATION 1] IMPLICIT SEQUENCE */
{
    IA5String givenName; /* IA5String */
    IA5String initial; /* IA5String */
    IA5String familyName; /* IA5String */
} Name;
AsnLen BEncNameContent PROTO((BUF_TYPE b, Name* v));
void BDecNameContent PROTO(( BUF_TYPE b, AsnTag tagId0, AsnLen elmtLen0,
Name* v, AsnLen* bytesDecoded, ENV_TYPE env));
void PrintName PROTO((FILE* f, Name* v, unsigned short int indent));
void FreeName PROTO((Name* v));


typedef IA5String Date; /* [APPLICATION 3] IMPLICIT IA5String */
#define BEncDateContent BEncIA5StringContent
#define BDecDateContent BDecIA5StringContent
#define PrintDate PrintIA5String
```

```
#define FreeDate FreeIA5String


typedef struct ChildInformation /* SET */
{
    struct Name* name; /* Name */
    Date dateOfBirth; /* [0] Date */
} ChildInformation;
AsnLen BEncChildInformationContent PROTO((BUF_TYPE b, ChildInformation* v));
void BDecChildInformationContent PROTO(( BUF_TYPE b, AsnTag tagId0,
AsnLen elmtLen0, ChildInformation* v, AsnLen* bytesDecoded, ENV_TYPE env));
void PrintChildInformation PROTO((FILE* f, ChildInformation* v,
unsigned short int indent));
void FreeChildInformation PROTO((ChildInformation* v));



typedef AsnList PersonnelRecordSeqOf; /* SEQUENCE OF ChildInformation */
AsnLen BEncPersonnelRecordSeqOfContent PROTO((BUF_TYPE b,
PersonnelRecordSeqOf* v));
void BDecPersonnelRecordSeqOfContent PROTO(( BUF_TYPE b, AsnTag tagId0,
AsnLen elmtLen0, PersonnelRecordSeqOf* v, AsnLen* bytesDecoded, ENV_TYPE env));
void PrintPersonnelRecordSeqOf PROTO((FILE* f, PersonnelRecordSeqOf* v,
unsigned short int indent));
void FreePersonnelRecordSeqOf PROTO((PersonnelRecordSeqOf* v));



typedef struct PersonnelRecord /* [APPLICATION 0] IMPLICIT SET */
{
    struct Name* name; /* Name */
    IA5String title; /* [0] IA5String */
    EmployeeNumber employeeNumber; /* EmployeeNumber */
    Date dateOfHire; /* [1] Date */
    struct Name* nameOfSpouse; /* [2] Name */
    PersonnelRecordSeqOf* children;
        /* [3] IMPLICIT PersonnelRecordSeqOf DEFAULT
         -- snacc warning: can't parse this value yet --{ } */
} PersonnelRecord;
AsnLen BEncPersonnelRecord PROTO((BUF_TYPE b, PersonnelRecord* v));
void BDecPersonnelRecord PROTO(( BUF_TYPE b, PersonnelRecord* result,
```

```
AsnLen* bytesDecoded, ENV_TYPE env));
AsnLen BEncPersonnelRecordContent PROTO((BUF_TYPE b, PersonnelRecord* v));
void BDecPersonnelRecordContent PROTO(( BUF_TYPE b, AsnTag tagId0,
AsnLen elmtLen0, PersonnelRecord* v, AsnLen* bytesDecoded, ENV_TYPE env));
void PrintPersonnelRecord PROTO((FILE* f, PersonnelRecord* v,
unsigned short int indent));
void FreePersonnelRecord PROTO((PersonnelRecord* v));

#endif /* conditional include of p_rec.h */
```

Snacc generated C file p_rec.c

```
/*
 *    p_rec.c
 *
 *    "P-REC" ASN.1 module encode/decode/print/free C src.
 *
 *    This file was generated by snacc on Sun Apr 11 02:17:21 1993
 *
 *    UBC snacc written by Mike Sample
 *
 *    NOTE: This is a machine generated file - editing not recommended
 */

#include "asn_incl.h"
#include "p_rec.h"

AsnLen
BEncNameContent PARAMS((b, v),
BUF_TYPE b _AND_
Name* v)
{
    AsnLen totalLen = 0;
    AsnLen itemLen;
    AsnLen listLen;
    void* component;

    itemLen = BEncIA5StringContent( b, (&v->familyName));
    itemLen += BEncDefLen( b, itemLen);
    itemLen += BEncTag1(b, UNIV, PRIM, 22);

    totalLen += itemLen;

    itemLen = BEncIA5StringContent( b, (&v->initial));
    itemLen += BEncDefLen( b, itemLen);
    itemLen += BEncTag1(b, UNIV, PRIM, 22);

    totalLen += itemLen;

    itemLen = BEncIA5StringContent( b, (&v->givenName));
    itemLen += BEncDefLen( b, itemLen);
    itemLen += BEncTag1(b, UNIV, PRIM, 22);
```

```
    totalLen += itemLen;

    return (totalLen);

}  /* BEncNameContent */

void
BDecNameContent PARAMS((b, tagId0, elmtLen0, v, bytesDecoded, env),
BUF_TYPE b _AND_
AsnTag tagId0 _AND_
AsnLen elmtLen0 _AND_
Name* v _AND_
AsnLen* bytesDecoded _AND_
ENV_TYPE env)
{
    int seqDone = FALSE;
    AsnLen totalElmtsLen1 = 0;
    AsnLen elmtLen1;
    AsnTag tagId1;
    int mandatoryElmtCount1 = 0;


    tagId1 = BDecTag(b, &totalElmtsLen1, env);

    if ((( tagId1 == MAKE_TAG_ID( UNIV, PRIM, IA5STRING_TAG_CODE)) ||
( tagId1 == MAKE_TAG_ID( UNIV, CONS, IA5STRING_TAG_CODE))))
    {
    elmtLen1 = BDecLen (b, &totalElmtsLen1, env);
    BDecIA5StringContent( b, tagId1, elmtLen1, (&v->givenName),
                          &totalElmtsLen1, env);
    tagId1 = BDecTag(b, &totalElmtsLen1, env);
    }
    else
        longjmp(env, -100);


    if ((( tagId1 == MAKE_TAG_ID( UNIV, PRIM, IA5STRING_TAG_CODE)) ||
( tagId1 == MAKE_TAG_ID( UNIV, CONS, IA5STRING_TAG_CODE))))
    {
    elmtLen1 = BDecLen (b, &totalElmtsLen1, env);
    BDecIA5StringContent( b, tagId1, elmtLen1, (&v->initial),
```

```
                        &totalElmtsLen1, env);
    tagId1 = BDecTag(b, &totalElmtsLen1, env);
    }
    else
        longjmp(env, -101);


    if ((( tagId1 == MAKE_TAG_ID( UNIV, PRIM, IA5STRING_TAG_CODE)) ||
( tagId1 == MAKE_TAG_ID( UNIV, CONS, IA5STRING_TAG_CODE))))
    {
    elmtLen1 = BDecLen (b, &totalElmtsLen1, env);
    BDecIA5StringContent( b, tagId1, elmtLen1, (&v->familyName),
                         &totalElmtsLen1, env);
        seqDone = TRUE;
        if ( elmtLen0 == INDEFINITE_LEN )
            BDecEoc(b, &totalElmtsLen1, env);
        else if (totalElmtsLen1 != elmtLen0)
            longjmp(env, -102);
    }
    else
        longjmp(env, -103);


    if (!seqDone)
        longjmp(env, -104);

    (*bytesDecoded) += totalElmtsLen1;
}  /* BDecNameContent */

void
PrintName PARAMS((f, v, indent),
FILE* f _AND_
Name* v _AND_
unsigned short int indent)
{
    if (v == NULL)
        return;

    fprintf(f,"{ -- SEQUENCE --\n");

    Indent(f, indent + stdIndentG);
```

```
    fprintf(f,"givenName ");
    PrintIA5String(f, (&v->givenName), indent + stdIndentG);
    fprintf(f, ",\n");
    Indent(f, indent + stdIndentG);
    fprintf(f,"initial ");
    PrintIA5String(f, (&v->initial), indent + stdIndentG);
    fprintf(f, ",\n");
    Indent(f, indent + stdIndentG);
    fprintf(f,"familyName ");
    PrintIA5String(f, (&v->familyName), indent + stdIndentG);
    fprintf(f,"\n");
    Indent(f, indent);
    fprintf(f,"}");
}  /* PrintName */

void
FreeName PARAMS((v),
Name* v)
{

    if (v == NULL)
        return;
    FreeIA5String((&v->givenName));

    FreeIA5String((&v->initial));

    FreeIA5String((&v->familyName));

}  /* FreeName */



AsnLen
BEncChildInformationContent PARAMS((b, v),
BUF_TYPE b _AND_
ChildInformation* v)
{
    AsnLen totalLen = 0;
    AsnLen itemLen;
    AsnLen listLen;
    void* component;
```

```
    BEncEocIfNec(b);
    itemLen = BEncDateContent( b, (&v->dateOfBirth));
    itemLen += BEncDefLen( b, itemLen);
    itemLen += BEncTag1(b, APPL, PRIM, 3);
    itemLen += BEncConsLen( b, itemLen);
    itemLen += BEncTag1(b, CNTX, CONS, 0);

    totalLen += itemLen;

    BEncEocIfNec(b);
    itemLen = BEncNameContent( b, (v->name));
    itemLen += BEncConsLen( b, itemLen);
    itemLen += BEncTag1(b, APPL, CONS, 1);

    totalLen += itemLen;

    return (totalLen);

}  /* BEncChildInformationContent */

void
BDecChildInformationContent PARAMS((b, tagId0, elmtLen0, v, bytesDecoded, env),
BUF_TYPE b _AND_
AsnTag tagId0 _AND_
AsnLen elmtLen0 _AND_
ChildInformation* v _AND_
AsnLen* bytesDecoded _AND_
ENV_TYPE env)
{
    int seqDone = FALSE;
    AsnLen totalElmtsLen1 = 0;
    AsnLen elmtLen1;
    AsnTag tagId1;
    int mandatoryElmtCount1 = 0;
    AsnLen totalElmtsLen2 = 0;
    AsnLen elmtLen2;
    AsnTag tagId2;


for ( ; (totalElmtsLen1 < elmtLen0) || (elmtLen0 == INDEFINITE_LEN);)
```

```
    {
    tagId1 = BDecTag(b, &totalElmtsLen1, env);

    if ( (tagId1 == EOC_TAG_ID) && (elmtLen0 == INDEFINITE_LEN))
    {
        BDEC_2ND_EOC_OCTET(b, &totalElmtsLen1, env)
        break; /* got EOC so can exit this SET's for loop*/
    }
    elmtLen1 = BDecLen (b, &totalElmtsLen1, env);
    switch(tagId1)
    {
       case(MAKE_TAG_ID( APPL, CONS, 1)):
    (v->name) = (Name*) Asn1Alloc(sizeof(Name));
    CheckAsn1Alloc((v->name), env);
    BDecNameContent( b, tagId1, elmtLen1, (v->name), &totalElmtsLen1, env);
    mandatoryElmtCount1++;
    break;

       case(MAKE_TAG_ID( CNTX, CONS, 0)):
    tagId2 = BDecTag(b, &totalElmtsLen1, env);
if ((tagId2 != MAKE_TAG_ID( APPL, PRIM, 3)) &&
   (tagId2 != MAKE_TAG_ID( APPL, CONS, 3)))
    {
         Asn1Error("Unexpected Tag\n");
         longjmp(env, -106);
    }

elmtLen2 = BDecLen (b, &totalElmtsLen1, env);
    BDecDateContent( b, tagId2, elmtLen2, (&v->dateOfBirth),
                    &totalElmtsLen1, env);
    if (elmtLen1 == INDEFINITE_LEN)
        BDecEoc(b, &totalElmtsLen1, env);
    mandatoryElmtCount1++;
    break;

    default:
        Asn1Error("BDecChildInformationContent: ERROR - Unexpected tag\
in SET\n");
        longjmp(env, -107);
        break;
        } /* end switch */
```

```
    } /* end for */
    if (mandatoryElmtCount1 != 2)
    {
        Asn1Error("BDecChildInformationContent: ERROR - non-optional elmt\
missing from SET\n");
        longjmp(env, -108);
    }
    (*bytesDecoded) += totalElmtsLen1;
}  /* BDecChildInformationContent */

void
PrintChildInformation PARAMS((f, v, indent),
FILE* f _AND_
ChildInformation* v _AND_
unsigned short int indent)
{
    if (v == NULL)
        return;

    fprintf(f,"{ -- SET --\n");

    Indent(f, indent + stdIndentG);
    PrintName(f, (v->name), indent + stdIndentG);
    fprintf(f, ",\n");
    Indent(f, indent + stdIndentG);
    fprintf(f,"dateOfBirth ");
    PrintDate(f, (&v->dateOfBirth), indent + stdIndentG);
    fprintf(f,"\n");
    Indent(f, indent);
    fprintf(f,"}");
}  /* PrintChildInformation */

void
FreeChildInformation PARAMS((v),
ChildInformation* v)
{

    if (v == NULL)
        return;
    FreeName((v->name));
    Asn1Free((v->name));
```

```
    FreeDate((&v->dateOfBirth));

}  /* FreeChildInformation */



AsnLen
BEncPersonnelRecordSeqOfContent PARAMS((b, v),
BUF_TYPE b _AND_
PersonnelRecordSeqOf* v)
{
    AsnLen totalLen = 0;
    AsnLen itemLen;
    AsnLen listLen;
    void* component;

        listLen = 0;
    FOR_EACH_LIST_ELMT_RVS( component, v)
    {
    BEncEocIfNec(b);
    itemLen = BEncChildInformationContent( b, component);
    itemLen += BEncConsLen( b, itemLen);
    itemLen += BEncTag1(b, UNIV, CONS, 17);

        listLen += itemLen;
    }
    return (listLen);

}  /* BEncPersonnelRecordSeqOfContent */

void
BDecPersonnelRecordSeqOfContent PARAMS((b, tagId0, elmtLen0, v,
bytesDecoded, env),
BUF_TYPE b _AND_
AsnTag tagId0 _AND_
AsnLen elmtLen0 _AND_
PersonnelRecordSeqOf* v _AND_
AsnLen* bytesDecoded _AND_
ENV_TYPE env)
```

```
{
    int seqDone = FALSE;
    AsnLen totalElmtsLen1 = 0;
    AsnLen elmtLen1;
    AsnTag tagId1;
    int mandatoryElmtCount1 = 0;


    for ( totalElmtsLen1 = 0; (totalElmtsLen1 < elmtLen0) ||
         (elmtLen0 == INDEFINITE_LEN);)
    {
        ChildInformation** tmpVar;
    tagId1 = BDecTag(b, &totalElmtsLen1, env);

    if ( (tagId1 == EOC_TAG_ID) && (elmtLen0 == INDEFINITE_LEN))
    {
        BDEC_2ND_EOC_OCTET(b, &totalElmtsLen1, env)
        break; /* got EOC so can exit this SET OF/SEQ OF's for loop*/
    }
    if ( (tagId1 == MAKE_TAG_ID( UNIV, CONS, SET_TAG_CODE)))
    {
        elmtLen1 = BDecLen (b, &totalElmtsLen1, env);
    tmpVar = (ChildInformation**) AsnListAppend(v);
    (*tmpVar) = (ChildInformation*) Asn1Alloc(sizeof(ChildInformation));
    CheckAsn1Alloc((*tmpVar), env);
    BDecChildInformationContent( b, tagId1, elmtLen1, (*tmpVar),
                                 &totalElmtsLen1, env);
    }  /* end of tag check if */
    else  /* wrong tag */
    {
         Asn1Error("Unexpected Tag\n");
         longjmp(env, -109);
    }
    } /* end of for */

    (*bytesDecoded) += totalElmtsLen1;
}  /* BDecPersonnelRecordSeqOfContent */

void
PrintPersonnelRecordSeqOf PARAMS((f, v, indent),
FILE* f _AND_
```

```
PersonnelRecordSeqOf* v _AND_
unsigned short int indent)
{
    ChildInformation* tmp;
    if (v == NULL)
        return;
    fprintf(f,"{ -- SEQUENCE OF -- \n");
    FOR_EACH_LIST_ELMT(tmp, v)
    {
        Indent(f, indent+ stdIndentG);
        PrintChildInformation(f, tmp, indent + stdIndentG);
        if (tmp != (ChildInformation*)LAST_LIST_ELMT(v))
            fprintf(f,",\n");
    }
    fprintf(f,"\n");
    Indent(f, indent);
    fprintf(f,"}");
}  /* PrintPersonnelRecordSeqOf */

void
FreePersonnelRecordSeqOf PARAMS((v),
PersonnelRecordSeqOf* v)
{

    AsnListNode* l;
    AsnListNode* tmp;
    if (v == NULL)
        return;
    for (l = FIRST_LIST_NODE(v); l != NULL; )
    {
        FreeChildInformation((l->data));
        tmp = l->next;
        Asn1Free(l->data);
        Asn1Free(l);
        l = tmp;
    }
}  /* FreePersonnelRecordSeqOf */
```

```
AsnLen BEncPersonnelRecord PARAMS((b, v),
BUF_TYPE b _AND_
PersonnelRecord* v)
{
    AsnLen l;
    BEncEocIfNec(b);
    l = BEncPersonnelRecordContent(b, v);
    l += BEncConsLen(b, l);
    l += BEncTag1(b, APPL, CONS, 0);
    return(l);
} /* BEncPersonnelRecord */

void BDecPersonnelRecord PARAMS((b, result, bytesDecoded, env),
BUF_TYPE b _AND_
PersonnelRecord* result _AND_
AsnLen* bytesDecoded _AND_
ENV_TYPE env)
{
    AsnTag tag;
    AsnLen elmtLen1;

    if ( ((tag = BDecTag(b, bytesDecoded, env)) !=
MAKE_TAG_ID(APPL, CONS, 0)))
    {
        Asn1Error("BDecPersonnelRecord: ERROR - wrong tag\n");
        longjmp(env, -110);
    }
    elmtLen1 = BDecLen(b, bytesDecoded, env);
    BDecPersonnelRecordContent(b, tag, elmtLen1, result, bytesDecoded, env);
}  /* BDecPersonnelRecord */

AsnLen
BEncPersonnelRecordContent PARAMS((b, v),
BUF_TYPE b _AND_
PersonnelRecord* v)
{
    AsnLen totalLen = 0;
    AsnLen itemLen;
    AsnLen listLen;
    void* component;
```

```
if (NOT_NULL((v->children)))
{
BEncEocIfNec(b);
itemLen = BEncPersonnelRecordSeqOfContent( b, (v->children));
itemLen += BEncConsLen( b, itemLen);
itemLen += BEncTag1(b, CNTX, CONS, 3);

totalLen += itemLen;
}

BEncEocIfNec(b);
BEncEocIfNec(b);
itemLen = BEncNameContent( b, (v->nameOfSpouse));
itemLen += BEncConsLen( b, itemLen);
itemLen += BEncTag1(b, APPL, CONS, 1);
itemLen += BEncConsLen( b, itemLen);
itemLen += BEncTag1(b, CNTX, CONS, 2);

totalLen += itemLen;

BEncEocIfNec(b);
itemLen = BEncDateContent( b, (&v->dateOfHire));
itemLen += BEncDefLen( b, itemLen);
itemLen += BEncTag1(b, APPL, PRIM, 3);
itemLen += BEncConsLen( b, itemLen);
itemLen += BEncTag1(b, CNTX, CONS, 1);

totalLen += itemLen;

itemLen = BEncEmployeeNumberContent( b, (&v->employeeNumber));
BEncDefLenTo127( b, itemLen);
itemLen++;
itemLen += BEncTag1(b, APPL, PRIM, 2);

totalLen += itemLen;

BEncEocIfNec(b);
itemLen = BEncIA5StringContent( b, (&v->title));
itemLen += BEncDefLen( b, itemLen);
itemLen += BEncTag1(b, UNIV, PRIM, 22);
itemLen += BEncConsLen( b, itemLen);
```

```
    itemLen += BEncTag1(b, CNTX, CONS, 0);

    totalLen += itemLen;

    BEncEocIfNec(b);
    itemLen = BEncNameContent( b, (v->name));
    itemLen += BEncConsLen( b, itemLen);
    itemLen += BEncTag1(b, APPL, CONS, 1);

    totalLen += itemLen;

    return (totalLen);

}  /* BEncPersonnelRecordContent */

void
BDecPersonnelRecordContent PARAMS((b, tagId0, elmtLen0, v, bytesDecoded, env),
BUF_TYPE b _AND_
AsnTag tagId0 _AND_
AsnLen elmtLen0 _AND_
PersonnelRecord* v _AND_
AsnLen* bytesDecoded _AND_
ENV_TYPE env)
{
    int seqDone = FALSE;
    AsnLen totalElmtsLen1 = 0;
    AsnLen elmtLen1;
    AsnTag tagId1;
    int mandatoryElmtCount1 = 0;
    AsnLen totalElmtsLen2 = 0;
    AsnLen elmtLen2;
    AsnTag tagId2;


for ( ; (totalElmtsLen1 < elmtLen0) || (elmtLen0 == INDEFINITE_LEN);)
    {
    tagId1 = BDecTag(b, &totalElmtsLen1, env);

    if ( (tagId1 == EOC_TAG_ID) && (elmtLen0 == INDEFINITE_LEN))
    {
        BDEC_2ND_EOC_OCTET(b, &totalElmtsLen1, env)
```

```
        break; /* got EOC so can exit this SET's for loop*/
    }
    elmtLen1 = BDecLen (b, &totalElmtsLen1, env);
    switch(tagId1)
    {
       case(MAKE_TAG_ID( APPL, CONS, 1)):
    (v->name) = (Name*) Asn1Alloc(sizeof(Name));
    CheckAsn1Alloc((v->name), env);
    BDecNameContent( b, tagId1, elmtLen1, (v->name), &totalElmtsLen1, env);
    mandatoryElmtCount1++;
    break;

       case(MAKE_TAG_ID( CNTX, CONS, 0)):
    tagId2 = BDecTag(b, &totalElmtsLen1, env);
if ((tagId2 != MAKE_TAG_ID( UNIV, PRIM, IA5STRING_TAG_CODE)) &&
   (tagId2 != MAKE_TAG_ID( UNIV, CONS, IA5STRING_TAG_CODE)))
    {
         Asn1Error("Unexpected Tag\n");
         longjmp(env, -111);
    }

elmtLen2 = BDecLen (b, &totalElmtsLen1, env);
    BDecIA5StringContent( b, tagId2, elmtLen2, (&v->title),
                          &totalElmtsLen1, env);
    if (elmtLen1 == INDEFINITE_LEN)
        BDecEoc(b, &totalElmtsLen1, env);
    mandatoryElmtCount1++;
    break;

       case(MAKE_TAG_ID( APPL, PRIM, 2)):
    BDecEmployeeNumberContent( b, tagId1, elmtLen1, (&v->employeeNumber),
                               &totalElmtsLen1, env);
    mandatoryElmtCount1++;
    break;

       case(MAKE_TAG_ID( CNTX, CONS, 1)):
    tagId2 = BDecTag(b, &totalElmtsLen1, env);
if ((tagId2 != MAKE_TAG_ID( APPL, PRIM, 3)) &&
   (tagId2 != MAKE_TAG_ID( APPL, CONS, 3)))
    {
         Asn1Error("Unexpected Tag\n");
```

```
        longjmp(env, -112);
    }

elmtLen2 = BDecLen (b, &totalElmtsLen1, env);
    BDecDateContent( b, tagId2, elmtLen2, (&v->dateOfHire),
                    &totalElmtsLen1, env);
    if (elmtLen1 == INDEFINITE_LEN)
        BDecEoc(b, &totalElmtsLen1, env);
    mandatoryElmtCount1++;
    break;

       case(MAKE_TAG_ID( CNTX, CONS, 2)):
if (BDecTag(b, &totalElmtsLen1, env) != MAKE_TAG_ID( APPL, CONS, 1))
    {
        Asn1Error("Unexpected Tag\n");
        longjmp(env, -113);
    }

elmtLen2 = BDecLen (b, &totalElmtsLen1, env);
    (v->nameOfSpouse) = (Name*) Asn1Alloc(sizeof(Name));
    CheckAsn1Alloc((v->nameOfSpouse), env);
    BDecNameContent( b, tagId2, elmtLen2, (v->nameOfSpouse),
                    &totalElmtsLen1, env);
    if (elmtLen1 == INDEFINITE_LEN)
        BDecEoc(b, &totalElmtsLen1, env);
    mandatoryElmtCount1++;
    break;

       case(MAKE_TAG_ID( CNTX, CONS, 3)):
    (v->children) = AsnListNew(sizeof(char*));
    CheckAsn1Alloc((v->children), env);
    BDecPersonnelRecordSeqOfContent( b, tagId1, elmtLen1, (v->children),
                                    &totalElmtsLen1, env);
    break;

    default:
        Asn1Error("BDecPersonnelRecordContent: ERROR - Unexpected tag
 in SET\n");
        longjmp(env, -114);
        break;
        } /* end switch */
```

```
    } /* end for */
    if (mandatoryElmtCount1 != 5)
    {
        Asn1Error("BDecPersonnelRecordContent: ERROR - non-optional elmt\
missing from SET\n");
        longjmp(env, -115);
    }
    (*bytesDecoded) += totalElmtsLen1;
}  /* BDecPersonnelRecordContent */

void
PrintPersonnelRecord PARAMS((f, v, indent),
FILE* f _AND_
PersonnelRecord* v _AND_
unsigned short int indent)
{
    if (v == NULL)
        return;

    fprintf(f,"{ -- SET --\n");

    Indent(f, indent + stdIndentG);
    PrintName(f, (v->name), indent + stdIndentG);
    fprintf(f, ",\n");
    Indent(f, indent + stdIndentG);
    fprintf(f,"title ");
    PrintIA5String(f, (&v->title), indent + stdIndentG);
    fprintf(f, ",\n");
    Indent(f, indent + stdIndentG);
    PrintEmployeeNumber(f, (&v->employeeNumber), indent + stdIndentG);
    fprintf(f, ",\n");
    Indent(f, indent + stdIndentG);
    fprintf(f,"dateOfHire ");
    PrintDate(f, (&v->dateOfHire), indent + stdIndentG);
    fprintf(f, ",\n");
    Indent(f, indent + stdIndentG);
    fprintf(f,"nameOfSpouse ");
    PrintName(f, (v->nameOfSpouse), indent + stdIndentG);
    if ( NOT_NULL((v->children)))
    {
    fprintf(f,",\n");
```

```
    Indent(f, indent + stdIndentG);
    fprintf(f,"children ");
    PrintPersonnelRecordSeqOf(f, (v->children), indent + stdIndentG);
    }
    fprintf(f,"\n");
    Indent(f, indent);
    fprintf(f,"}");
}  /* PrintPersonnelRecord */

void
FreePersonnelRecord PARAMS((v),
PersonnelRecord* v)
{

    if (v == NULL)
        return;
    FreeName((v->name));
    Asn1Free((v->name));

    FreeIA5String((&v->title));

    FreeEmployeeNumber((&v->employeeNumber));

    FreeDate((&v->dateOfHire));

    FreeName((v->nameOfSpouse));
    Asn1Free((v->nameOfSpouse));

    if ( NOT_NULL((v->children)))
    {
    FreePersonnelRecordSeqOf((v->children));
    Asn1Free((v->children));
    }

}  /* FreePersonnelRecord */
```

Appendix B

C++ Example

This appendix contains an example of the C++ code snacc will generate for an ASN.1 data structure. Snacc generated two files for the PersonnelRecord ASN.1 data structure, p_rec.C and p_rec.h.

Snacc generated C++ file p_rec.h

```
//
// p_rec.h - class definitions for ASN.1 module P-REC
//
//   This file was generated by snacc on Sun Apr 11 03:17:42 1993
//   UBC snacc by Mike Sample
//   NOTE: this is a machine generated file - editing not recommended
//

#ifndef _p_rec_h_
#define _p_rec_h_


// forward declarations

class EmployeeNumber;
class Name;
class Date;
class ChildInformation;
class PersonnelRecordSeqOf;
class PersonnelRecord;

 /* [APPLICATION 2] IMPLICIT INTEGER */
class EmployeeNumber : public AsnInt
{
    public:
      EmployeeNumber(): AsnInt() {}
      EmployeeNumber(int i): AsnInt(i) {}
      AsnType* Clone() { return new EmployeeNumber; }
      AsnLen BEnc(BUF_TYPE b)
      {
          AsnLen l;
          l = BEncContent(b);
    l += BEncConsLen( b, l);
          l += BEncTag1(b, APPL, PRIM, 2);
          return(l);
      }

      void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env)
      {
          AsnTag tag;
          AsnLen elmtLen1;
```

```
          if ( ((tag = BDecTag(b, bytesDecoded, env)) !=
MAKE_TAG_ID(APPL, PRIM, 2)))
          {
              Asn1Error("EmployeeNumber::BDec: ERROR - wrong tag\n");
              longjmp(env, -100);
          }
          elmtLen1 = BDecLen(b, bytesDecoded, env);
          BDecContent(b, tag, elmtLen1, bytesDecoded, env);
      }

      int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded)
      {
           bytesEncoded = BEnc(b);
           return(!b.WriteError());
      }

      int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded)
      {
          ENV_TYPE env;
          int val;

          bytesDecoded = 0;
          if ((val = setjmp(env)) == 0 )
          {
               BDec(b, bytesDecoded, env);
               return(!b.ReadError());
          }
          else
              return(FALSE);
      }

};


class Name : public AsnType
{
    public:
      IA5String givenName;
      IA5String initial;
      IA5String familyName;
```

```
Name() {}
void Print(ostream& os);
AsnType* Clone() { return new Name; }
AsnLen BEnc(BUF_TYPE b)
{
    AsnLen l;
    l = BEncContent(b);
    l += BEncConsLen(b, l);
    l += BEncTag1(b, APPL, CONS, 1);
    return(l);
}

void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env)
{
    AsnTag tag;
    AsnLen elmtLen1;

    if ( (tag = BDecTag(b, bytesDecoded, env)) !=
         MAKE_TAG_ID(APPL, CONS, 1))
    {
        Asn1Error("Name::BDec: ERROR - wrong tag\n");
        longjmp(env, -101);
    }
    elmtLen1 = BDecLen(b, bytesDecoded, env);
    BDecContent(b, tag, elmtLen1, bytesDecoded, env);
}

AsnLen BEncContent(BUF_TYPE b);
void   BDecContent(BUF_TYPE b, AsnTag tag, AsnLen elmtLen,
                   AsnLen& bytesDecoded, ENV_TYPE env);

int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded)
{
     bytesEncoded = BEnc(b);
     return(!b.WriteError());
}

int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded)
{
    ENV_TYPE env;
```

```
          int val;

          bytesDecoded = 0;
          if ((val = setjmp(env)) == 0 )
          {
               BDec(b, bytesDecoded, env);
               return(!b.ReadError());
          }
          else
              return(FALSE);
      }

};


 /* [APPLICATION 3] IMPLICIT IA5String */
class Date : public IA5String
{
    public:
      Date(): IA5String() {}
      Date(const char* str): IA5String(str) {}
      Date(const char* str, const unsigned long int len): IA5String(str,len) {}
      Date(const IA5String& o): IA5String(o) {}
      Date& operator=(const Date& o) {ReSet(o); return *this;}
      Date& operator=(const char* str) {ReSet(str); return *this;}
      AsnType* Clone() { return new Date; }
      AsnLen BEnc(BUF_TYPE b)
      {
          AsnLen l;
          l = BEncContent(b);
    l += BEncDefLen( b, l);
          l += BEncTag1(b, APPL, PRIM, 3);
          return(l);
      }

      void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env)
      {
          AsnTag tag;
          AsnLen elmtLen1;

          if ( ((tag = BDecTag(b, bytesDecoded, env)) !=
```

```
MAKE_TAG_ID(APPL, PRIM, 3))&&
         (tag != MAKE_TAG_ID(APPL, CONS, 3)))
          {
              Asn1Error("Date::BDec: ERROR - wrong tag\n");
              longjmp(env, -106);
          }
          elmtLen1 = BDecLen(b, bytesDecoded, env);
          BDecContent(b, tag, elmtLen1, bytesDecoded, env);
      }

      int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded)
      {
           bytesEncoded = BEnc(b);
           return(!b.WriteError());
      }

      int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded)
      {
          ENV_TYPE env;
          int val;

          bytesDecoded = 0;
          if ((val = setjmp(env)) == 0 )
          {
               BDec(b, bytesDecoded, env);
               return(!b.ReadError());
          }
          else
              return(FALSE);
      }

};


class ChildInformation : public AsnType
{
    public:
      Name* name;
      Date dateOfBirth;

      ChildInformation() {}
```

```
        void Print(ostream& os);
        AsnType* Clone() { return new ChildInformation; }
        AsnLen BEnc(BUF_TYPE b)
        {
            AsnLen l;
            l = BEncContent(b);
            l += BEncConsLen(b, l);
            l += BEncTag1(b, UNIV, CONS, SET_TAG_CODE);
            return(l);
        }

        void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env)
        {
            AsnTag tag;
            AsnLen elmtLen1;

            if ( (tag = BDecTag(b, bytesDecoded, env)) !=
MAKE_TAG_ID(UNIV, CONS, SET_TAG_CODE))
            {
                Asn1Error("ChildInformation::BDec: ERROR - wrong tag\n");
                longjmp(env, -107);
            }
            elmtLen1 = BDecLen(b, bytesDecoded, env);
            BDecContent(b, tag, elmtLen1, bytesDecoded, env);
        }

        AsnLen BEncContent(BUF_TYPE b);
        void   BDecContent(BUF_TYPE b, AsnTag tag, AsnLen elmtLen,
                           AsnLen& bytesDecoded, ENV_TYPE env);

        int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded)
        {
             bytesEncoded = BEnc(b);
             return(!b.WriteError());
        }

        int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded)
        {
            ENV_TYPE env;
            int val;
```

```
          bytesDecoded = 0;
          if ((val = setjmp(env)) == 0 )
          {
               BDec(b, bytesDecoded, env);
               return(!b.ReadError());
          }
          else
              return(FALSE);
      }

};


class PersonnelRecordSeqOf : public AsnType
{
  protected:
    unsigned long int count;
    struct AsnListElmt
    {
        struct AsnListElmt* next;
        struct AsnListElmt* prev;
        ChildInformation* elmt;
    } *first, *curr, *last;

  public:
    void Print(ostream& os);
      AsnType* Clone() { return new PersonnelRecordSeqOf; }
    PersonnelRecordSeqOf () { count = 0; first = curr = last = NULL; }
    void SetCurrElmt(unsigned long int index);
    unsigned long int GetCurrElmtIndex();
    void SetCurrToFirst() { curr = first; }
    void SetCurrToLast()  { curr = last; }
    // reading member fcns
    int Count() { return count; }
    ChildInformation* First() { if (count > 0) return(first->elmt) ;
                                else return(NULL); }
    ChildInformation* Last()  { if (count > 0) return(last->elmt) ;
                                else return(NULL); }
    ChildInformation* Curr()  { if (curr != NULL) return(curr->elmt) ;
                                else return(NULL); }
    ChildInformation* Next()  { if ((curr != NULL)&& (curr->next != NULL))
```

```
                              return(curr->next->elmt) ; else return(NULL); }
ChildInformation* Prev()  { if ((curr != NULL)&& (curr->prev != NULL))
                              return(curr->prev->elmt) ; else return(NULL); }

// routines that move the curr elmt
ChildInformation* GoNext() { if (curr != NULL) curr = curr->next;
                               return(Curr()); }
ChildInformation* GoPrev() { if (curr != NULL) curr = curr->prev;
                               return(Curr()); }

// write & alloc fcns - returns new elmt
ChildInformation* Append();  // add elmt to end of list
ChildInformation* Prepend(); // add elmt to beginning of list
ChildInformation* InsertBefore(); //insert elmt before current elmt
ChildInformation* InsertAfter(); //insert elmt after current elmt

// write & alloc & copy - returns list after copying elmt
PersonnelRecordSeqOf& AppendCopy( ChildInformation& elmt);
PersonnelRecordSeqOf& PrependCopy( ChildInformation& elmt);
PersonnelRecordSeqOf& InsertBeforeAndCopy( ChildInformation& elmt);
PersonnelRecordSeqOf& InsertAfterAndCopy( ChildInformation& elmt);

// encode and decode routines
AsnLen BEnc(BUF_TYPE b)
{
    AsnLen l;
    l = BEncContent(b);
    l += BEncConsLen(b, l);
    l += BEncTag1(b, UNIV, CONS, SEQ_TAG_CODE);
    return(l);
}

void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env)
{
    AsnTag tag;
    AsnLen elmtLen1;

    if ( (tag = BDecTag(b, bytesDecoded, env)) !=
         MAKE_TAG_ID(UNIV, CONS, SEQ_TAG_CODE))
   {
        Asn1Error("PersonnelRecordSeqOf::BDec: ERROR - wrong tag\n");
```

```
            longjmp(env, -111);
        }
        elmtLen1 = BDecLen(b, bytesDecoded, env);
         BDecContent(b, tag, elmtLen1, bytesDecoded, env);
    }

    AsnLen BEncContent(BUF_TYPE b);
    void   BDecContent(BUF_TYPE b, AsnTag tag, AsnLen elmtLen,
                       AsnLen& bytesDecoded, ENV_TYPE env);

    PDU_MEMBER_MACROS
};


class PersonnelRecord : public AsnType
{
    public:
      Name* name;
      IA5String title;
      EmployeeNumber employeeNumber;
      Date dateOfHire;
      Name* nameOfSpouse;
      PersonnelRecordSeqOf* children;

      PersonnelRecord()
      {
          /* init optional/default elements to NULL */
          children = NULL;
      }

      void Print(ostream& os);
      AsnType* Clone() { return new PersonnelRecord; }
      AsnLen BEnc(BUF_TYPE b)
      {
          AsnLen l;
          l = BEncContent(b);
          l += BEncConsLen(b, l);
          l += BEncTag1(b, APPL, CONS, 0);
          return(l);
      }
```

```
        void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env)
        {
            AsnTag tag;
            AsnLen elmtLen1;

            if ( (tag = BDecTag(b, bytesDecoded, env)) !=
MAKE_TAG_ID(APPL, CONS, 0))
            {
                Asn1Error("PersonnelRecord::BDec: ERROR - wrong tag\n");
                longjmp(env, -113);
            }
            elmtLen1 = BDecLen(b, bytesDecoded, env);
            BDecContent(b, tag, elmtLen1, bytesDecoded, env);
        }

        AsnLen BEncContent(BUF_TYPE b);
        void   BDecContent(BUF_TYPE b, AsnTag tag, AsnLen elmtLen,
                           AsnLen& bytesDecoded, ENV_TYPE env);

        int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded)
        {
             bytesEncoded = BEnc(b);
             return(!b.WriteError());
        }

        int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded)
        {
            ENV_TYPE env;
            int val;

            bytesDecoded = 0;
            if ((val = setjmp(env)) == 0 )
            {
                 BDec(b, bytesDecoded, env);
                 return(!b.ReadError());
            }
            else
                return(FALSE);
        }

};
```

```
// externs for value defs

#endif /* conditional include of p_rec.h */
```

Snacc generated C++ file p_rec.C

```
//
// p_rec.C - class member functions for ASN.1 module P-REC
//
//   This file was generated by snacc on Sun Apr 11 03:17:42 1993
//   UBC snacc written by Mike Sample
//   NOTE: this is a machine generated file - editing not recommended
//


#include "asn_incl.h"
#include "p_rec.h"


// value defs



AsnLen
Name::BEncContent(BUF_TYPE b)
{
    AsnLen totalLen = 0;
    AsnLen l;

    l = familyName.BEncContent(b);
    l += BEncDefLen( b, l);
    l += BEncTag1(b, UNIV, PRIM, IA5STRING_TAG_CODE);
    totalLen += l;

    l = initial.BEncContent(b);
    l += BEncDefLen( b, l);
    l += BEncTag1(b, UNIV, PRIM, IA5STRING_TAG_CODE);
    totalLen += l;

    l = givenName.BEncContent(b);
    l += BEncDefLen( b, l);
    l += BEncTag1(b, UNIV, PRIM, IA5STRING_TAG_CODE);
    totalLen += l;

    return(totalLen);
} /* Name::BEncContent */
```

```
void
Name::BDecContent(BUF_TYPE b, AsnTag tag0, AsnLen elmtLen0,
AsnLen& bytesDecoded, ENV_TYPE env)
{
    AsnTag tag1;
    AsnLen seqBytesDecoded = 0;
    AsnLen elmtLen1;
    tag1 = BDecTag(b, seqBytesDecoded, env);

    if (( tag1 == MAKE_TAG_ID( UNIV, PRIM, IA5STRING_TAG_CODE)) ||
        ( tag1 == MAKE_TAG_ID( UNIV, CONS, IA5STRING_TAG_CODE)))
    {
        elmtLen1 = BDecLen(b, seqBytesDecoded, env);
        givenName.BDecContent(b, tag1, elmtLen1, seqBytesDecoded, env);
        tag1 = BDecTag(b, seqBytesDecoded, env);
    }
    else
    {
        Asn1Error("ERROR - SEQUENCE is missing non-optional elmt.\n");
        longjmp(env, -102);
    }

    if (( tag1 == MAKE_TAG_ID( UNIV, PRIM, IA5STRING_TAG_CODE)) ||
        ( tag1 == MAKE_TAG_ID( UNIV, CONS, IA5STRING_TAG_CODE)))
    {
        elmtLen1 = BDecLen(b, seqBytesDecoded, env);
        initial.BDecContent(b, tag1, elmtLen1, seqBytesDecoded, env);
        tag1 = BDecTag(b, seqBytesDecoded, env);
    }
    else
    {
        Asn1Error("ERROR - SEQUENCE is missing non-optional elmt.\n");
        longjmp(env, -103);
    }

    if (( tag1 == MAKE_TAG_ID( UNIV, PRIM, IA5STRING_TAG_CODE)) ||
        ( tag1 == MAKE_TAG_ID( UNIV, CONS, IA5STRING_TAG_CODE)))
    {
        elmtLen1 = BDecLen(b, seqBytesDecoded, env);
        familyName.BDecContent(b, tag1, elmtLen1, seqBytesDecoded, env);
```

```
    }
    else
    {
        Asn1Error("ERROR - SEQUENCE is missing non-optional elmt.\n");
        longjmp(env, -104);
    }

    bytesDecoded += seqBytesDecoded;
    if ( elmtLen0 == INDEFINITE_LEN )
    {
        BDecEoc(b, bytesDecoded, env);
        return;
    }
    else if (seqBytesDecoded != elmtLen0)
    {
        Asn1Error("ERROR - Length discrepancy on sequence.\n");
        longjmp(env, -105);
    }
    else
        return;
} /* Name::BDecContent */

void
Name::Print(ostream& os)
{
    os << "{  -- SEQUENCE --" << endl;
    indentG += stdIndentG;
    Indent(os, indentG);
    os << "givenName ";
    os << givenName;
    os << "," << endl;
    Indent(os, indentG);
    os << "initial ";
    os << initial;
    os << "," << endl;
    Indent(os, indentG);
    os << "familyName ";
    os << familyName;
    os << endl;
    indentG -= stdIndentG;
    Indent(os, indentG);
```

```
    os << "}";
}  /* Name::Print */


AsnLen
ChildInformation::BEncContent(BUF_TYPE b)
{
    AsnLen totalLen = 0;
    AsnLen l;

    BEncEocIfNec(b);
    l = dateOfBirth.BEncContent(b);
    l += BEncDefLen( b, l);
    l += BEncTag1(b, APPL, PRIM, 3);
    l += BEncConsLen( b, l);
    l += BEncTag1(b, CNTX, CONS, 0);
    totalLen += l;

    BEncEocIfNec(b);
    l = name->BEncContent(b);
    l += BEncConsLen( b, l);
    l += BEncTag1(b, APPL, CONS, 1);
    totalLen += l;

    return(totalLen);
} /* ChildInformation::BEncContent */


void
ChildInformation::BDecContent(BUF_TYPE b, AsnTag tag0, AsnLen elmtLen0,
AsnLen& bytesDecoded, ENV_TYPE env)
{
    AsnTag tag1;
    AsnLen setBytesDecoded = 0;
    unsigned int mandatoryElmtsDecoded = 0;
    AsnLen elmtLen1;
    AsnLen elmtLen2;

    for( ; (setBytesDecoded < elmtLen0) || (elmtLen0 == INDEFINITE_LEN);)
    {
        tag1 = BDecTag(b, setBytesDecoded, env);
```

```
        if ((elmtLen0 == INDEFINITE_LEN) && (tag1 == EOC_TAG_ID))
        {
            BDEC_2ND_EOC_OCTET(b, setBytesDecoded, env)
            break; /* exit for loop */
        }
        elmtLen1 = BDecLen (b, setBytesDecoded, env);
        switch(tag1)
        {
          case(MAKE_TAG_ID( APPL, CONS, 1)):
            name = new Name;
        name->BDecContent(b, tag1, elmtLen1, setBytesDecoded, env);
            mandatoryElmtsDecoded++;
            break;

          case(MAKE_TAG_ID( CNTX, CONS, 0)):
            tag1 = BDecTag(b, setBytesDecoded, env);
            if ((tag1 != MAKE_TAG_ID( APPL, PRIM, 3)) &&
               (tag1 != MAKE_TAG_ID( APPL, CONS, 3)))
            {
                 Asn1Error("Unexpected Tag\n");
                 longjmp(env, -108);
            }

            elmtLen2 = BDecLen (b, setBytesDecoded, env);
        dateOfBirth.BDecContent(b, tag1, elmtLen2, setBytesDecoded, env);
            if (elmtLen1 == INDEFINITE_LEN)
                BDecEoc(b, setBytesDecoded, env);

            mandatoryElmtsDecoded++;
            break;

           default:
             Asn1Error("Unexpected Tag on SET elmt.\n");
             longjmp(env, -109);
        }  /* end switch */
    }  /* end for loop */
    bytesDecoded += setBytesDecoded;
    if (mandatoryElmtsDecoded != 2)
    {
         Asn1Error("ERROR - non-optional SET element missing.\n");
```

```
            longjmp(env, -110);
        }
} /* ChildInformation::BDecContent */

void
ChildInformation::Print(ostream& os)
{
    os << "{  -- SET --" << endl;
    indentG += stdIndentG;
    Indent(os, indentG);
    os <<  *name;
    os << "," << endl;
    Indent(os, indentG);
    os << "dateOfBirth ";
    os << dateOfBirth;
    os << endl;
    indentG -= stdIndentG;
    Indent(os, indentG);
    os << "}";
}  /* ChildInformation - operator<< */


void PersonnelRecordSeqOf::Print(ostream& os)
{
    os << "{ -- SEQUENCE/SET OF -- " << endl;
    indentG += stdIndentG;
    SetCurrToFirst();
    for (; Curr() != NULL; GoNext())
    {
        Indent(os, indentG);
        os << *Curr();
        if ( Curr() != Last())
            os << ",";
        os << endl;
    }
    indentG -= stdIndentG;
    Indent(os, indentG);
    os << "}";
}   /* Print */
```

```
void  PersonnelRecordSeqOf::SetCurrElmt(unsigned long int index)
{
    unsigned long int i;
    curr = first;
    for( i = 0; (i < (count-1)) && (i < index); i++)
    {
        curr = curr->next;
    }
}  /*  PersonnelRecordSeqOf::SetCurrElmt */


unsigned long int  PersonnelRecordSeqOf::GetCurrElmtIndex()
{
    unsigned long int i;
    struct AsnListElmt* tmp;
    if (curr != NULL)
    {
        for( i = 0, tmp = first; tmp != NULL; i++)
        {
            if (tmp == curr)
                return(i);
            else
                tmp = tmp->next;
        }
    }
    return(count);
}  /*  PersonnelRecordSeqOf::GetCurrElmtIndex */


// alloc new list elmt, put at end of list
//  and return the component type
ChildInformation* PersonnelRecordSeqOf::Append()
{
    struct AsnListElmt* newElmt;
    newElmt  = new struct AsnListElmt;
    newElmt->elmt  = new ChildInformation;
    newElmt->next = NULL;
    if( last == NULL )
    {
newElmt->prev = NULL;
first = last  = newElmt;
```

```
    }
    else
    {
newElmt->prev = last;
        last->next    = newElmt;
last          = newElmt;
    }
    count++;
    return( newElmt->elmt );
} /*  PersonnelRecordSeqOf::Append */


// alloc new list elmt, put at begining of list
//  and return the component type
ChildInformation*  PersonnelRecordSeqOf::Prepend()
{
    struct AsnListElmt* newElmt;
    newElmt  = new struct AsnListElmt;
    newElmt->elmt = new ChildInformation;
    newElmt->prev = NULL;
    if( first == NULL )
    {
newElmt->next = NULL;
first = last  = newElmt;
    }
    else
    {
newElmt->next = first;
        first->prev   = newElmt;
first         = newElmt;
    }
    count++;
    return( newElmt->elmt );
} /*  PersonnelRecordSeqOf::Prepend */


// alloc new list elmt, insert it before the
// current element and return the component type
// if the current element is null, the new element
// is placed at the beginning of the list.
ChildInformation*  PersonnelRecordSeqOf::InsertBefore()
```

```
{
    struct AsnListElmt* newElmt;
    newElmt  = new struct AsnListElmt;
    newElmt->elmt = new ChildInformation;
    if (curr == NULL)
    {
        newElmt->next = first;
        newElmt->prev = NULL;
        first = newElmt;
        if (last == NULL)
            last = newElmt;
    }
    else
    {
newElmt->next = curr;
        newElmt->prev = curr->prev;
        curr->prev = newElmt;
        if (curr == first)
            first = newElmt;
        else
            newElmt->prev->next = curr;
    }
    count++;
    return( newElmt->elmt );
} /*  PersonnelRecordSeqOf::InsertBefore */


// alloc new list elmt, insert it after the
// current element and return the component type
// if the current element is null, the new element
// is placed at the end of the list.
ChildInformation* PersonnelRecordSeqOf::InsertAfter()
{
    struct AsnListElmt* newElmt;
    newElmt  = new struct AsnListElmt;
    newElmt->elmt = new ChildInformation;
    if (curr == NULL)
    {
        newElmt->prev = last;
        newElmt->next = NULL;
        last = newElmt;
```

```
        if (first == NULL)
            first = newElmt;
    }
    else
    {
newElmt->prev = curr;
        newElmt->next = curr->next;
        curr->next = newElmt;
        if (curr == last)
            last = newElmt;
        else
            curr->next->prev = newElmt;
    }
    count++;
    return( newElmt->elmt );
} /*  PersonnelRecordSeqOf::InsertAfter */
PersonnelRecordSeqOf&  PersonnelRecordSeqOf::AppendCopy(
ChildInformation& elmt)
{
    struct AsnListElmt* newElmt;
    newElmt  = new struct AsnListElmt;
    newElmt->elmt = new ChildInformation;
    *newElmt->elmt = elmt;
    newElmt->next = NULL;
    if( last == NULL )
    {
newElmt->prev = NULL;
first = last  = newElmt;
    }
    else
    {
 newElmt->prev = last;
        last->next    = newElmt;
 last          = newElmt;
    }
    count++;
    return(*this);
} /* AppendCopy */


PersonnelRecordSeqOf&  PersonnelRecordSeqOf::PrependCopy(
```

```
ChildInformation& elmt)
{
    struct AsnListElmt* newElmt;
    newElmt  = new struct AsnListElmt;
    newElmt->elmt = new ChildInformation;
    *newElmt->elmt = elmt;
    newElmt->prev = NULL;
    if( first == NULL )
    {
newElmt->next = NULL;
first = last  = newElmt;
    }
    else
    {
newElmt->next = first;
        first->prev   = newElmt;
first         = newElmt;
    }
    count++;
    return(*this);
} /*  PersonnelRecordSeqOf::PrependCopy */


// alloc new list elmt, insert it before the
// current element, copy the given elmt into the new elmt
// and return the component type.
// if the current element is null, the new element
// is placed at the beginning of the list.
PersonnelRecordSeqOf&  PersonnelRecordSeqOf::InsertBeforeAndCopy(
ChildInformation& elmt)
{
    struct AsnListElmt* newElmt;

    newElmt  = new struct AsnListElmt;
    newElmt->elmt = new ChildInformation;
    *newElmt->elmt = elmt;

    if (curr == NULL)
    {
        newElmt->next = first;
        newElmt->prev = NULL;
```

```
        first = newElmt;
        if (last == NULL)
            last = newElmt;
    }
    else
    {
newElmt->next = curr;
        newElmt->prev = curr->prev;
        curr->prev = newElmt;
        if (curr == first)
            first = newElmt;
        else
            newElmt->prev->next = curr;
    }
    count++;
    return( *this );
} /* PersonnelRecordSeqOf::InsertBeforeAndCopy */
// alloc new list elmt, insert it after the
// current element, copy given elmt in to new elmt
//  and return the component type
// if the current element is null, the new element
// is placed at the end of the list.
PersonnelRecordSeqOf&  PersonnelRecordSeqOf::InsertAfterAndCopy(
ChildInformation& elmt)
{
    struct AsnListElmt* newElmt;

    newElmt  = new struct AsnListElmt;
    newElmt->elmt = new ChildInformation;
    *newElmt->elmt = elmt;
    if (curr == NULL)
    {
        newElmt->prev = last;
        newElmt->next = NULL;
        last = newElmt;
        if (first == NULL)
            first = newElmt;
    }
    else
    {
newElmt->prev = curr;
```

```
        newElmt->next = curr->next;
        curr->next = newElmt;
        if (curr == last)
            last = newElmt;
        else
            curr->next->prev = newElmt;
    }
    count++;
    return( *this );
} /*  PersonnelRecordSeqOf::InsertAfterAndCopy */
AsnLen PersonnelRecordSeqOf::BEncContent(BUF_TYPE b)
{
    struct AsnListElmt* currElmt;
    AsnLen elmtLen;
    AsnLen totalLen = 0;
    for (currElmt = last; currElmt != NULL; currElmt = currElmt->prev)
    {
    BEncEocIfNec(b);
        elmtLen = currElmt->elmt->BEncContent(b);
    elmtLen += BEncConsLen( b, elmtLen);
    elmtLen += BEncTag1(b, UNIV, CONS, SET_TAG_CODE);
        totalLen += elmtLen;
    }
    return(totalLen);
} /* PersonnelRecordSeqOf::BEncContent */


void  PersonnelRecordSeqOf::BDecContent( BUF_TYPE b, AsnTag tag0,
AsnLen elmtLen0, AsnLen& bytesDecoded, ENV_TYPE env)
{
    ChildInformation* listElmt;
    AsnTag tag1;
    AsnLen listBytesDecoded = 0;
    AsnLen elmtLen1;

    while ((listBytesDecoded < elmtLen0) || (elmtLen0 == INDEFINITE_LEN))
    {
        tag1 = BDecTag(b, listBytesDecoded, env);
        if ((tag1 == EOC_TAG_ID) && (elmtLen0 == INDEFINITE_LEN))
        {
            BDEC_2ND_EOC_OCTET(b, listBytesDecoded, env);
```

```
            break;
        }
        if ((tag1 != MAKE_TAG_ID( UNIV, CONS, SET_TAG_CODE)))
        {
            Asn1Error("Unexpected Tag\n");
            longjmp(env, -112);
        }

        elmtLen1 = BDecLen (b, listBytesDecoded, env);
        listElmt = Append();
        listElmt->BDecContent(b, tag1, elmtLen1, listBytesDecoded, env);
    }

    bytesDecoded += listBytesDecoded;
}  /*  PersonnelRecordSeqOf::BDecContent */


AsnLen
PersonnelRecord::BEncContent(BUF_TYPE b)
{
    AsnLen totalLen = 0;
    AsnLen l;

    if (NOT_NULL(children))
    {
    BEncEocIfNec(b);
    l = children->BEncContent(b);
    l += BEncConsLen( b, l);
    l += BEncTag1(b, CNTX, CONS, 3);
    totalLen += l;
    }

    BEncEocIfNec(b);
    BEncEocIfNec(b);
    l = nameOfSpouse->BEncContent(b);
    l += BEncConsLen( b, l);
    l += BEncTag1(b, APPL, CONS, 1);
    l += BEncConsLen( b, l);
    l += BEncTag1(b, CNTX, CONS, 2);
    totalLen += l;
```

```
    BEncEocIfNec(b);
    l = dateOfHire.BEncContent(b);
    l += BEncDefLen( b, l);
    l += BEncTag1(b, APPL, PRIM, 3);
    l += BEncConsLen( b, l);
    l += BEncTag1(b, CNTX, CONS, 1);
    totalLen += l;

    l = employeeNumber.BEncContent(b);
    BEncDefLenTo127( b, l);
    l++;
    l += BEncTag1(b, APPL, PRIM, 2);
    totalLen += l;

    BEncEocIfNec(b);
    l = title.BEncContent(b);
    l += BEncDefLen( b, l);
    l += BEncTag1(b, UNIV, PRIM, IA5STRING_TAG_CODE);
    l += BEncConsLen( b, l);
    l += BEncTag1(b, CNTX, CONS, 0);
    totalLen += l;

    BEncEocIfNec(b);
    l = name->BEncContent(b);
    l += BEncConsLen( b, l);
    l += BEncTag1(b, APPL, CONS, 1);
    totalLen += l;

    return(totalLen);
} /* PersonnelRecord::BEncContent */


void
PersonnelRecord::BDecContent(BUF_TYPE b, AsnTag tag0, AsnLen elmtLen0,
AsnLen& bytesDecoded, ENV_TYPE env)
{
    AsnTag tag1;
    AsnLen setBytesDecoded = 0;
    unsigned int mandatoryElmtsDecoded = 0;
    AsnLen elmtLen1;
    AsnLen elmtLen2;
```

```
for( ; (setBytesDecoded < elmtLen0) || (elmtLen0 == INDEFINITE_LEN);)
{
    tag1 = BDecTag(b, setBytesDecoded, env);

    if ((elmtLen0 == INDEFINITE_LEN) && (tag1 == EOC_TAG_ID))
    {
        BDEC_2ND_EOC_OCTET(b, setBytesDecoded, env)
        break; /* exit for loop */
    }
    elmtLen1 = BDecLen (b, setBytesDecoded, env);
    switch(tag1)
    {
      case(MAKE_TAG_ID( APPL, CONS, 1)):
        name = new Name;
    name->BDecContent(b, tag1, elmtLen1, setBytesDecoded, env);
        mandatoryElmtsDecoded++;
        break;

      case(MAKE_TAG_ID( CNTX, CONS, 0)):
        tag1 = BDecTag(b, setBytesDecoded, env);
        if ((tag1 != MAKE_TAG_ID( UNIV, PRIM, IA5STRING_TAG_CODE)) &&
           (tag1 != MAKE_TAG_ID( UNIV, CONS, IA5STRING_TAG_CODE)))
        {
             Asn1Error("Unexpected Tag\n");
             longjmp(env, -114);
        }

        elmtLen2 = BDecLen (b, setBytesDecoded, env);
    title.BDecContent(b, tag1, elmtLen2, setBytesDecoded, env);
        if (elmtLen1 == INDEFINITE_LEN)
            BDecEoc(b, setBytesDecoded, env);

        mandatoryElmtsDecoded++;
        break;

      case(MAKE_TAG_ID( APPL, PRIM, 2)):
    employeeNumber.BDecContent(b, tag1, elmtLen1, setBytesDecoded, env);
        mandatoryElmtsDecoded++;
        break;
```

```
  case(MAKE_TAG_ID( CNTX, CONS, 1)):
    tag1 = BDecTag(b, setBytesDecoded, env);
    if ((tag1 != MAKE_TAG_ID( APPL, PRIM, 3)) &&
       (tag1 != MAKE_TAG_ID( APPL, CONS, 3)))
    {
         Asn1Error("Unexpected Tag\n");
         longjmp(env, -115);
    }

    elmtLen2 = BDecLen (b, setBytesDecoded, env);
dateOfHire.BDecContent(b, tag1, elmtLen2, setBytesDecoded, env);
    if (elmtLen1 == INDEFINITE_LEN)
        BDecEoc(b, setBytesDecoded, env);

    mandatoryElmtsDecoded++;
    break;

  case(MAKE_TAG_ID( CNTX, CONS, 2)):
    tag1 = BDecTag(b, setBytesDecoded, env);
    if (tag1 != MAKE_TAG_ID( APPL, CONS, 1))
    {
         Asn1Error("Unexpected Tag\n");
         longjmp(env, -116);
    }

    elmtLen2 = BDecLen (b, setBytesDecoded, env);
    nameOfSpouse = new Name;
nameOfSpouse->BDecContent(b, tag1, elmtLen2, setBytesDecoded, env);
    if (elmtLen1 == INDEFINITE_LEN)
        BDecEoc(b, setBytesDecoded, env);

    mandatoryElmtsDecoded++;
    break;

  case(MAKE_TAG_ID( CNTX, CONS, 3)):
    children = new PersonnelRecordSeqOf;
children->BDecContent(b, tag1, elmtLen1, setBytesDecoded, env);
    break;

   default:
     Asn1Error("Unexpected Tag on SET elmt.\n");
```

```
            longjmp(env, -117);
        }  /* end switch */
    }  /* end for loop */
    bytesDecoded += setBytesDecoded;
    if (mandatoryElmtsDecoded != 5)
    {
         Asn1Error("ERROR - non-optional SET element missing.\n");
         longjmp(env, -118);
    }
} /* PersonnelRecord::BDecContent */

void
PersonnelRecord::Print(ostream& os)
{
    os << "{  -- SET --" << endl;
    indentG += stdIndentG;
    Indent(os, indentG);
    os <<  *name;
    os << "," << endl;
    Indent(os, indentG);
    os << "title ";
    os << title;
    os << "," << endl;
    Indent(os, indentG);
    os << employeeNumber;
    os << "," << endl;
    Indent(os, indentG);
    os << "dateOfHire ";
    os << dateOfHire;
    os << "," << endl;
    Indent(os, indentG);
    os << "nameOfSpouse ";
    os <<  *nameOfSpouse;
    if (NOT_NULL(children))
    {
    os << ","<< endl;
    Indent(os, indentG);
    os << "children ";
    os <<  *children;
    }
    os << endl;
```

```
    indentG -= stdIndentG;
    Indent(os, indentG);
    os << "}";
}  /* PersonnelRecord - operator<< */
```

Appendix C

The Type Table Data Structure

This appendix contains the definitions of the type table data structure.

```
--
--  TBL types describe ASN.1 data structures.
--  These can be used in generic, interpretive encoders/decoders.
--
--  The type table is defined in ASN.1 to provide a simple (BER)
--  file format for tables. It also allows tables be sent over a
--  network for dynamic re-configuration of encoders/decoders.
--
--  MS 93

TBL DEFINITIONS ::=
BEGIN


-- imports nothing
-- exports nothing

TBL ::= --snacc isPdu:"TRUE" -- SEQUENCE
{
   totalNumModules  INTEGER,  -- these totals can help allocation
   totalNumTypeDefs INTEGER,  -- when decoding (ie use arrays)
   totalNumTypes    INTEGER,
   totalNumTags     INTEGER,
   totalNumStrings  INTEGER,
   totalLenStrings  INTEGER,
   modules SEQUENCE OF TBLModule
}

TBLModule ::= SEQUENCE
{
    name     [0] IMPLICIT PrintableString,
```

```
    id       [1] IMPLICIT OBJECT IDENTIFIER OPTIONAL,
    typeDefs [2] IMPLICIT SEQUENCE OF TBLTypeDef
}

TBLTypeDef ::= SEQUENCE
{
   typeDefId TBLTypeDefId,
   typeName  PrintableString OPTIONAL,
   type      TBLType
}

TBLType ::= SEQUENCE
{
   typeId    [0] IMPLICIT TBLTypeId,
   optional  [1] IMPLICIT BOOLEAN,
   tagList   [2] IMPLICIT SEQUENCE OF TBLTag OPTIONAL,
   content   [3] TBLTypeContent,
   fieldName [4] IMPLICIT PrintableString OPTIONAL
}

TBLTypeContent ::= CHOICE
{
   primType [0] IMPLICIT NULL,
   elmts    [1] IMPLICIT SEQUENCE OF TBLType,
   typeRef  [2] IMPLICIT TBLTypeRef
}

TBLTypeRef ::= SEQUENCE
{
    typeDef TBLTypeDefId,
    implicit BOOLEAN
}

TBLTypeId ::= ENUMERATED
{
    tbl-boolean(0),
    tbl-integer(1),
    tbl-bitstring(2),
    tbl-octetstring(3),
    tbl-null(4),
    tbl-oid(5),
```

```
    tbl-real(6),
    tbl-enumerated(7),
    tbl-sequence(8),
    tbl-set(9),
    tbl-sequenceof(10),
    tbl-setof(11),
    tbl-choice(12),
    tbl-typeref(13)
}

TBLTypeDefId ::= INTEGER

TBLTag ::= SEQUENCE
{
    tclass TBLTagClass,
    code INTEGER (0..MAX)
}

TBLTagClass ::=  ENUMERATED { universal(0), application(1),
                              context(2), private(3)}

END
```

Bibliography

[ASN184] CCITT, "Recommendation X.409, Presentation Transfer Syntax and Notation", Data Communications Networks Message Handling Systems, Red Book, Volume VIII, Fascicle VIII.7, pages 62–93, Omnicom, 115 Park St., S.E., Vienna, VA 22180 USA, Oct. 1985.

[ASN188] CCITT, "Recommendation X.208, Specification of Abstract Syntax Notation One (ASN. 1)", Data Communications Networks Open systems Interconnection (OSI) Model and Notation, Service Defintion, Blue Book, volume VIII, Fascicle VIII.4, pages 57–130, Omnicom, 115 Park St., S.E., Vienna, VA 22180 USA, Nov. 1989.

[BER88] CCITT, "Recommendation X.209, Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)", Data Communications Networks Open systems Interconnection (OSI) Model and Notation, Service Defintion, Blue Book, Volume VIII, Fascicle VIII.4, pages 130–151, Omnicom, 115 Park St., S.E., Vienna, VA 22180 USA, Nov. 1989.

[Bochman89] G. von Bochman and M. Deslouriers, "Combining ASN1 Support with the LOTOS Language", Proceedings of the International Symposium on Protocol Specification, Testing, and Verification IX, North Holland Publishers, 1989.

[Caneschi87] F. Caneschi and E. Merelli, "An Architecture for an ASN.1 Encoder/Decoder", Computer Networks and ISDN Systems, 14:297–303, 1987.

[Cardoso92] Artur Cardoso and Eduardo Tovar, "Defining More Efficient Transfer Syntax for Application Layer PDUs in Field Bus Applications", Sigcomm Computer Communication Review, 22(3):98–105, Jul. 1992.

[CDER92] ISO/IEC, "DIS 8825-3: Specification of ASN.1 Encoding Rules - Part 3: Distinguished and Canonical Encoding Rules", Jun. 1992.

[Clark89] David D. Clark, Van Jacobson, John Romkey and Howard Salwen, "An Analysis of TCP Processing Overhead", IEEE Communications Magazine, pages 23–29, Jun. 1989.

[Clark90] David D. Clark and David L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols", Sigcomm '90, 20(4):200–208, Sep. 1990.

[DeShon86] Annette L. DeSchon, "A Survey of Data Representation Standards (RFC 971)", Network Information Center, SRI International, Jan. 1986.

[DS88] CCITT, "Recommendation X.500, OSI:Specification of the Distributed Directory System", Data Communication Networks, Blue Book, Volume VIII, Fascicle VIII.8, pages 131–151, Omnicom, 115 Park St., S.E., Vienna, VA 22180 USA, Nov. 1989.

[Gaudette89] Phillip Gaudette, Steve Trus and Sarah Collins, "The Free Value Tool for ASN.1", National Institute of Standards and Technology, Feb. 1989.

[GRU90] MIPS Computer Inc., "GETRUSAGE(2-BSD)", RISC/os Reference Manual, Aug. 1990.

[PIX90] MIPS Computer Inc., "PIXIE(1-SysV)", RISC/os Reference Manual, Aug. 1990.

[Herlihy82] M. Herlihy and B. Liskov, "A Value Transmission Method For Abstract Data Types", ACM Transactions on Programming Languages and Systems, 4(4):527–551, Oct. 1982.

[Huitema89] Christian Huitema and Assem Doghri, "Defining Faster Transfer Syntaxes for the OSI Presentation Layer", Sigcomm Computer Communication Review, 19(5):44–55, Oct. 1989.

[Huitema92] Christian Huitema and Ghislain Chave, "Measuring the Performances of an ASN.1 Compiler", Upper Layer Protocols, Architectures and Applications, pages 99–112, May 1992.

[Kernighan88] Brian W. Kernighan and Dennis M. Ritchie, "The C Programming Language, 2nd Edition", Prentice-Hall, 1988.

[Kohl92] John Kohl and B. Clifford Neuman, "Internet-draft: The Kerberos Network Authentication Service (v5)", Sep. 1992

[Kong90] Mike Kong, "Network Computing System Reference Manual", Prentice-Hall, Inc., 1990.

[MHS88] CCITT, Recommendations X.400-X.420, Data Communication Networks Message Handling Systems, Blue Book, Volume VIII, Fascicle VIII.7, Omnicom, 115 Park St., S.E., Vienna, VA 22180 USA, Nov. 1989.

[MMS87] ISO/IEC, "9506: Manufacturing Message Specification (MMS)", Jun. 1987.

[NBS83] National Bureau of Standards, "Specification for Message Format for Computer Based Message Systems", (also published as RFC 841), Federal Information Processing Standards Publication 98, Jan. 1983.

[Neufeld90] Gerald Neufeld and Yeuli Yang, "An ASN.1 to C Compiler", IEEE Transactions on Software Engineering, 16(10):1209–1220, Oct. 1990.

[Neufeld92-1] Gerald Neufeld and Son Vuong, "An Overview of ASN.1", IEEE Networks and ISDN Systems, 23(5):393–415, Feb. 1992.

[Neufeld92-2] Gerald Neufeld, Barry Brachman, Murray Goldberg and Duncan Stickings, "The EAN X.500 Directory Service", Internetworking: Research and Experience, 3:55–81, 1992.

[OSF/RPC90] Open Software Foundation Inc. (OSF), "OSF/DCE Remote Procedure Call in a Distributed Computing Environment", 1990.

[OSI88] CCITT, "Recommendation X.200, Reference Model of Open Systems Interconnection for CCITT Applications", Data Communications Networks Open Systems Interconnection (OSI) Model and Notation, Service Defintion, Blue Book, Volume VIII, Fascile VIII.4, pages 3–56, Omnicom, 115 Park St., S.E., Vienna, VA 22180 USA, Nov. 1989.

[Partridge89] Craig Partridge and Marshall T. Rose, "A Comparison of External Data Formats", IFIP '89: Message Handling Systems and Distributed Applications, pages 233–245, Elsevier Science Publishers B.V. (North Holland), 1989.

[PER91] ISO/IEC, "8825-2, 1st CD: Specification of ASN.1 Encoding Rules - Part 2: Packed Encoding Rules", Oct. 1991.

[PER92] ISO/IEC, "8825-2, 2nd CD: Specification of ASN.1 Encoding Rules - Part 2: Packed Encoding Rules", Jun. 1992.

[Pimentel88] J.R. Pimentel, "Efficient Encoding of Application Layer PDUs for Fieldbus Networks", Sigcomm Computer Communication Review, 18(3):14–44, May 1988.

[Pope84] Arthur Pope, "Encoding CCITT X.409 Presentation Transfer Syntax", Sigcomm Computer Communication Review, 14(4):4–10, Oct. 1984.

[Postel80] J. Postel, "Internet Message Protocol (RFC 759)", Network Information Center, SRI International, Aug. 1980.

[Prasad93] Raja Prasad and Ulloa Gonzalo, "A Simple Encoder for Fieldbus Applications", Sigcomm Computer Communication Review, 23(1):34–44, Jan. 1993.

[ROS88] CCITT, "Remote Operations: Model, Notation and Service Definition", Data Communications Networks Open Systems Interconnection (OSI) Model and Notation, Service Defintion, Blue Book, Volume VIII, Fascicle VIII.4, pages 465–502, Omnicom, 115 Park St., S.E., Vienna, VA 22180 USA, Nov. 1989.

[Rose90] Marshall T. Rose, "ISODE, The ISO Development Environment: User Manual", Wollongong Group, 1129 San Antonio Rd., Palo Alto, California, USA, Feb. 1990.

[Sample93-1] Michael Sample, "Snacc 1.0: A High Performance ASN.1 to C/C++ Compiler (User Manual)". University of British Columbia, Feb. 1993.

[Sample93-2] Michael Sample and Gerald Neufeld, "Implementing Efficient Encoders and Decoders for Network Data Representations", IEEE INFOCOM '93 Proceedings, Volume 3, pages 1144–1153, Mar. 1993.

[snmp90] M. Rose and K. McCloghrie, "Structure and Identification of Management Information for TCP/IP-based Internets (RFC 1155)", Network Information Center, SRI International, May 1990.

[Steedman90] Douglass Steedman, "ASN.1, The Tutorial and Reference", Technology Appraisals Ltd., 1990.

[Steiner88] J. Steiner, C. Neuman, and J. Schiller, "Kerberos: An Authentication Service for Open Network Systems", Proceedings of the USENIX Winter Conference, Feb. 1988.

[Stroustrup91] Bjarne Stroustrup, "The C++ Programming Language", 2nd Edition, Addison-Wesley Publishing Co., 1991.

[Sun87] Sun Microsystems Inc., "XDR: External Data Representation Standard (RFC 1014)", Network Information Center, SRI International, Jun. 1987.

[Thomesse87] J.P. Thomesse and J.L. Delcuvellerie, "FIP: A Field Bus Proposal", NBS Workshop on Factory Communications, Gaithersburg, Md., March 1987.

[White89] James E. White, "ASN.1 and ROS: The Impact of X.400 on OSI", IEEE Journal on Selected Areas in Communications, 7(7):1060–1072, Sep. 1989.

[Xerox81] XEROX Corporation, "Courier: The Remote Procedure Call Protocol", xsis-038112, Dec. 1981.

[Yang88] Yueli Yang, "ASN.1-C Compiler for Automatic Protocol Implementation", Master's thesis, University of British Columbia, Oct. 1988.

[Z39.50] ANSI, "Z39.50/V2D3", May 1991.

[Zahn90] Lisa Zahn, "Network Computing Architecture", Prentice-Hall, Inc., 1990.