IN-61-CR
75 77
P-39

# EXPERIENCE WITH ABSTRACT NOTATION ONE

(NASA-CR-188090) EXPERIENCE WITH ABSTRACT NOTATION ONE (Houston Univ.) 39 p CSCL 09B

N91-21741

Unclas
G3/61 0007597

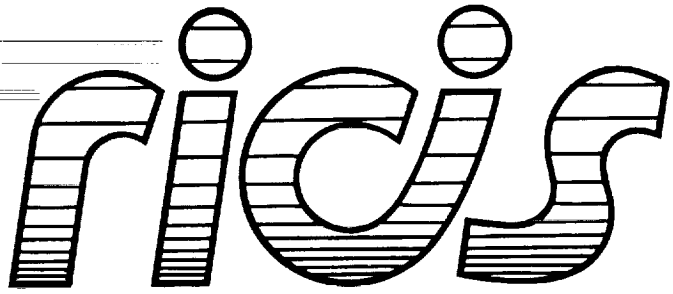
James D. Harvey
Alfred C. Weaver

*Digital Technology*

July 1990

Cooperative Agreement NCC 9-16
Research Activity No. SE.31

NASA Johnson Space Center
Engineering Directorate
Flight Data Systems Division




*Research Institute for Computing and Information Systems*
*University of Houston - Clear Lake*

**T·E·C·H·N·I·C·A·L R·E·P·O·R·T**

# *The RICIS Concept*

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

# EXPERIENCE WITH ABSTRACT NOTATION ONE

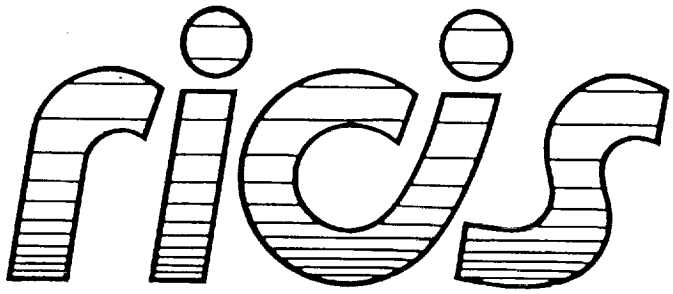
James D. Harvey
Alfred C. Weaver

*Digital Technology*

July 1990

Cooperative Agreement NCC 9-16
Research Activity No. SE.31

NASA Johnson Space Center
Engineering Directorate
Flight Data Systems Division



*Research Institute for Computing and Information Systems*
*University of Houston - Clear Lake*

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

## Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by James D. Harvey, Alfred C. Weaver and Digital Technology. Dr. George Collins, Associate Professor of Computer Systems Design, served as RICIS technical representative for this activity.

Funding has been provided by the Engineering Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Frank W. Miller, of the Systems Development Branch, Flight Data Systems Division, Engineering Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the authors and should not be interpreted as representative of the official policies, either express or implied, of NASA or the United States Government.

# ABSTRACT

The development of computer science has produced a vast number of machine architectures, programming languages, and compiler technologies. The cross product of these three characteristics defines the spectrum of previous and present data representation methodologies. With regard to computer networks, the uniqueness of these methodologies presents an obstacle when disparate host environments are to be interconnected. Interoperability within a heterogeneous network relies upon the establishment of data representation commonality.

The International Standards Organization (ISO) is currently developing the Abstract Syntax Notation One standard (ASN.1) and the Basic Encoding Rules standard (BER) that collectively address this problem. When used within the Presentation Layer of the Open Systems Interconnection Reference Model, these two standards provide the data representation commonality required to facilitate interoperability. This paper describes the details of a compiler that was built to automate the use of ASN.1 and BER. From this experience, insights into both standards are given and potential problems relating to this development effort are discussed.

# 1. Network Heterogeneity

The development of computer science has produced a vast number of disparate machine architectures, programming languages, and compiler technologies. The cross product of these three characteristics defines the spectrum of previous and present data representation methodologies. Although at one time the uniqueness of each technique provided vendors with a convenient and desirable means to monopolize their customers, the proliferation of computers and the increasing maturity of distributed processing have now obscured any of the previous advantages associated with this incompatibility. Nevertheless, established manufacturers are unwilling to abandon their investment in their own unique data representation schemes. Thus when computer networks interconnect these heterogeneous systems, a solution to the problem of data transfer across incompatible host environments must be provided.

## 1.1 Architectural Considerations

As a simple illustration of how hardware architecture influences this problem, consider three different techniques used to represent integer values: sign magnitude, diminished radix complement (one's complement), and radix complement (two's complement). Although most contemporary architectures now use two's complement, there are architectures currently in use that do not.[1] Consequently, if integer data were to be exchanged between a one's complement machine and a two's complement machine, each computer would have a different interpretation of the "same" negative values. Other data types also suffer from this representation disparity; perhaps the most obvious is floating point numbers.

1. The CDC 6600 is a one's complement architecture.

To complicate these architectural considerations even further, characteristics such as word alignment, byte ordering, and addressability may also present problems. Moreover, the influence of hardware architecture extends beyond just that of the CPU. The difference between the ASCII and EBCDIC character sets indicates that the architecture of peripherals also bears significance. The interaction and interdependencies that exist between all of these characteristics suggest that the influence of hardware architecture on the problem of disparate data representations is broad.

## 1.2 Languages and Compilers

The differences in conditional expression evaluation that occur within the C and Ada programming languages demonstrate the manner in which programming languages can contribute to this problem. In all C implementations a conditional expression is **true** if it evaluates to any non-zero value; it is **false** if it evaluates to zero precisely. Although the C programming language does not explicitly provide a boolean type, most C programmers use an integer data type to represent these kinds of values. In Ada, however, a boolean type *is* predefined as an enumerated type in package STANDARD. By virtue of its enumerated form, the only requirement relating to the representation of its values is that the representation of **false** be numerically less than the representation of **true**. Thus, each individual Ada compiler determines the exact nature of how boolean values are represented. Since Ada treats conditional expressions as boolean values, the evaluation of such expressions may clearly vary. In this respect, if the same application program were to be implemented in two different programming languages, it is feasible for two values that are intended to represent the "same" condition to be different. In fact, in certain cases it is more precise to attribute this phenomenon to the differences between compiler implementations than to the difference between programming language definitions.[2] Consequently, applications written in

the "same" programming language may experience miscommunication.

## 1.3 Solving the Problem

With regard to network heterogeneity, the disparate data representation problem is clear. Since different host environments may possess different methods of representing data, the interoperability of these machines can not be achieved solely by the establishment of a reliable connection. At some point during the communication, a transformation to and from each host machine's native representation must be performed. This function could be carried out in one of two ways: either (1) each host environment must be cognizant of the representation characteristics of each other host environment with which it wishes to communicate, and therefore each host must perform a potentially different transformation for every host-to-host combination, or (2) a common method of data representation must be established whereupon each host would be responsible for the single transformation between this standard method and its own native representation. This latter approach, which is certainly the more desirable alternative, captures the intent of the Abstract Syntax Notation One and the associated Basic Encoding Rules.

# 2. ASN.1

The ASN.1 standard[1] (ISO 8824) defines a language used to describe data values and data types. It is typically used by Application Layer protocols to define the types of their Application Protocol Data Units (APDUs). For example, protocols such as File Transfer, Access, and Management (FTAM) and Virtual Terminal (VT) use ASN.1. However, it is important to note

2. MicroSoft C compilers treat character values as signed quantities by default. Consequently, all character values are sign extended during type conversions. Lattice C compilers, however, consider all character values to be unsigned by default and do not sign extend.

that the use of ASN.1 is not necessarily restricted to the Application Layer. Theoretically, it could be used to define the Protocol Data Units (PDUs) of any layer.

In the OSI nomenclature, ASN.1 provides the means to define an **abstract syntax.** To understand this concept, it is useful to consider the term literally. In the formal, academic sense, a *syntax* defines the legal sentential forms of a language. Within the context of communications, one usually thinks of a syntax as defining the bit patterns used to represent data flowing across a medium. An *abstract* syntax does not define these bit patterns, but rather it establishes a framework whereupon these bit patterns, called a **transfer syntax,** can be created. The manner in which they are created depends upon the encoding rules that are applied.

## 2.1 Simple Types and Structured Types

There are two classes of ASN.1 data types that enable a user to describe the individual data units of a protocol, **simple** and **structured.** Simple types correspond to the primitive, atomic types typically found within most programming languages. The ASN.1 simple types include: booleans, integers, bit strings, character strings, and various other forms.[3] Structured types are identified by various ways in which existing ASN.1 types may be logically grouped or logically partitioned. ASN.1 defines five methods of formulating these structured types:

1. given an ordered list of existing types, a value can be formed as an ordered sequence of values, one from each of the existing types; the collection of all possible values obtained in this way is a new type;

3. Real types, enumerated types and subtypes are defined within a draft addendum and are therefore not currently part of the official ASN.1 standard. However, their inclusion with ISO 8824 is eminent.

2. given a list of distinct existing types, a value can be formed as an unordered set of values, one from each of the existing types; the collection of all possible values obtained in this way is a new type;

3. given a single existing type, a value can be formed as a ordered sequence of or unordered set of zero, one or more values of the existing type; the infinite colection of all possible values obtained in this way is a new type;

4. given a list of distinct types, a value can be chosen from any one of them; the set of all possible values obtained in this way is a new type;

5. given a type, a new type can be formed as a subset of it by using some structure or order relationship among the values;

These five methods correspond to the ASN.1 structured types: sequence, set, sequence-of or set-of, choice and subtypes, respectively.

## 2.2 Tags

Every data type that is defined using ASN.1 has an assigned tag. The value of this tag is either predefined by ASN.1 implicitly or defined by the user of the notation explicitly. The encoding rules always carry the tag of an ASN.1 type, whether explicit or implicit, with any representation of a value of the type. The purpose of this tag is to enable the distinct type of the represented value to be deduced during the decoding process. There are four classes of tags: universal, application, context-specific, and private. Universal tags are assigned exclusively by ISO 8824; application tags are assigned to types by other standards; private tags are never assigned by ISO Standards or CCITT Recommendations and are therefore enterprise specific, and context-specific tags may be assigned within any use of ASN.1 for the purpose disambiguating the type according to the context in which it is used.

## 2.3 An ASN.1 Example

To familiarize ourselves with ASN.1, let us consider a very simple, hypothetical set of data communication requirements relating to the NASA Space Station. From these requirements, we will define our own protocol using ASN.1.

One of the data communication requirements of the Space Station might involve the transfer of work requests and the transfer of corresponding results. The scheduling of a particular scientific experiment and the reporting of the results obtained from its execution are a perfect example of this kind of information exchange. The developement of a protocol that meets the needs of this communication problem requires that two PDUs be defined, *Request* and *Result*.

We will call this new application protocol the Work Management Protocol (WMP). In our ASN.1 definition of WMP we will recognize the following requirements:

1. A work request value should provide a textual description of the work to be performed.
2. A work request value should provide an indication of when the work is to be attempted. However, the specification of this element should be optional. If the requesting entity chooses to omit the specification of this value, then the work is to be performed at the receivers convenience.
3. A work request value should provide an indication of who is to attempt the work; this will entail the specification of zero or more individuals.[4] Furthermore, it is important that the user be given the ability to associate an authority hierarchy with the individuals that are specified. In other words, the manner in which these individuals are expressed should

4. In the case where no individuals are specified, the requested work may be carried out by whom ever is available at the time it is scheduled to commence.

reflect a "chain of command".

4. A result value should indicate more than just success or failure. It should also indicate the names of the individual(s) who attempted to perform the task. This may differ from the original request. The contents of a result value should also vary according to the outcome. If a work request is executed successfully, the time of completion should be given. If a request was not performed, an explanation of why it failed would be desirable.

5. Each work request value should possess an indication of its relative importance (priority).

6. A mechanism to associate each result value with its original request should be provided, otherwise the request and result transfers would have to occur in lockstep.

Considering these requirements, the ASN.1 definition of WMP might look as follows:

```
Wmp DEFINITIONS ::= BEGIN

   Request ::= SET {
      assigned-to Participants OPTIONAL,
      start-time UTCTime OPTIONAL,
      id-number [0] INTEGER,
      importance INTEGER {background(0), normal(1), urgent(2)},
      description IA5String
   }

   Result ::= SEQUENCE {
      Participants,
      id-number INTEGER,
      CHOICE {
         time-of-completion UTCTime,
         reason-for-failure IA5String
      }
   }

   Participants ::= SEQUENCE OF IA5String
     -- listed in order of decreasing authority

END
```

**Figure 1. An ASN.1 Module**

The block of text in Figure 1 represents a **module** definition. The module construct represents the ASN.1 way in which related type and value definitions may be logically grouped. Every module definition begins with a statement of the form:

<module name> DEFINITIONS ::= BEGIN

and ends with the keyword END. For a module corresponding to an international standard, it is recommended that its corresponding module name be of the form:

ISOxxxx-yyyy

where xxxx is the number of the international standard and yyyy is a suitable acronym.

Syntactically, the use of ASN.1 is somewhat flexible; the layout of the notation is not considered significant. Likewise, indenting is permitted and in fact encouraged. Its proper use will dramatically increase the readability of the notation. ASN.1 *is*, however, case sensitive. All keywords such as DEFINITIONS, BEGIN, END, SEQUENCE, and INTEGER must appear in upper case. Certain classes of identifiers must also begin with either upper or lower case letters specifically. For example, a module name must always begin with an upper case letter; all of its subsequent letters may be expressed in either upper or lower case, and hyphens may appear between any two.

This particular module definition consists of three **type assignments**; each defines a new ASN.1 data type and establishes a **type reference** (or name) that can be used to designate the type. All type references must begin with an upper case letter. The first type assignment defines the type reference *Request* as a SET of three elements. The second and third assignments define the *Result* type reference and *Participants* type reference as a SEQUENCE and a SEQUENCE OF, respectively.

The ASN.1 sequence is a constructor notation used to model an *ordered* collection of

variables whose number is known and modest and whose types may differ. To think of this in programming language terms, a sequence data type may be viewed as the ASN.1 equivalent of a Pascal record or a structure declaration in C. An ASN.1 set is identical to a sequence except for the property that the order of the elements within a set is not considered significant.[5] The ASN.1 sequence-of type is identical to the sequence type except for the fact that all of the elements of a sequence-of must be of the same type and the number of these elements is unbounded.

# 3. BER

The Basic Encoding Rules standard[2] (ISO 8825) defines a specific technique for encoding data. To use our previous OSI terminology, it defines a mapping from the abstract syntax, defined by ASN.1, into the transfer syntax. A BER encoding is represented as a sequence of octets. These octets are partitioned into 3-tuples, indicated in Figure 2:

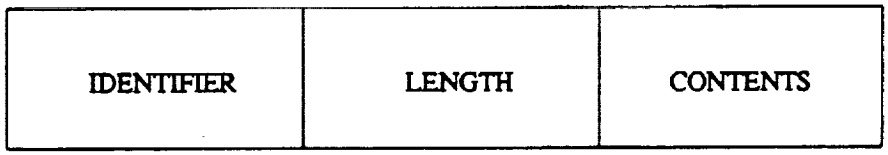



**Figure 2.** Encoding Format

The **identifier octet(s)** contain information regarding the type and form of the encoded data value. A **primitive** form indicates that the **contents octets** contain a direct representation of the data value. A **constructed** form indicates that the contents octets contain another embedded encoding. The **length octet(s)** determine the end of an encoding. They indicate how many contents octets represent the encoded data, or, if that this number is unknown, then they indicate

5. We have used the set type in this example to demonstrate its existence within ASN.1 and to called attention to the importance of explicit tagging. Note that due to the hidden inefficiencies associated with the decoding of set types in general, which is discussed later, we do not recommend that they be used.

that the contents octets are delimited by a reserved bit pattern called an **end of contents** (EOC) sequence. The contents octets contain a direct representation of the data value or another nested identifier-length-contents sequence.

### 3.0.1 The Identifier Octet(s)

The identifier octets denote the tag and form of an encoded data value. An identifier octet sequence can assume two formats. Which format to use in a given case depends upon the magnitude of the tag number. If the tag number falls within the range 0..30 inclusively, then the corresponding binary value may be placed within a five bit field, thus enabling the entire identifier information to fit into one octet. If, on the other hand, the tag number exceeds 30, additional octets must be used. Figure 24 indicates the layout of the single octet format.

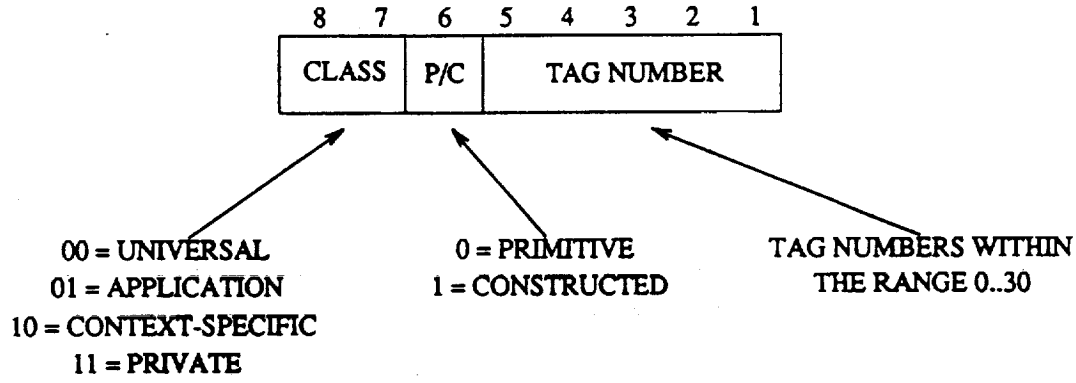



**Figure 3.** Identifier Octet (low tag number)

As illustrated, three information fields are contained within the identifier octet. The meaning of each is as follows:

1. **Class:** Bits 8-7 indicate the tag class of the encoded data value.

2. **Form:** Bit 6 indicates whether the encoded value is primitive or constructed. A primitive form indicates that the contents octets contain a direct representation of the data value. A

constructed form indicates that contents octets are the complete encoding of another data value. In other words, the subsequent contents octets contain embedded identifier-length-contents sequences.

3. **Tag Number:** For tag numbers falling within the range 0 to 30 inclusive, bits 5-1 of the leading (and only) octet designate this value. For tag numbers greater than 30, bits 5 through 1 contain the value 11111. This reserved bit pattern indicates that an extension of additional octets is required to hold the tag number. Bits 7-1 of each subsequent octet whose 8 bit is set to 1, up to and including the first octet whose 8 bit is set to zero, will be concatenated to form the binary tag number value. Figure 25 graphically illustrates this.

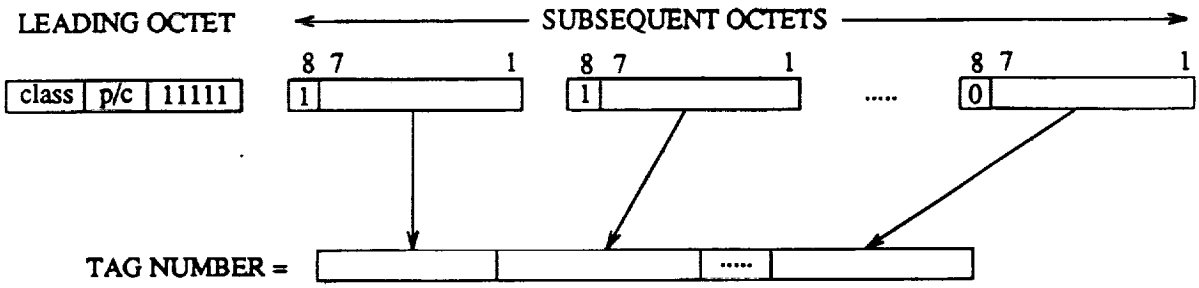



**Figure 4.** Identifier Octet (extended tag number)

### 3.0.2 The Length Octet(s)

The purpose of the length octets is to determine the end of an encoding. They explicitly indicate how many contents octets have been used in the encoding, or, alternatively, they signify that this length is unknown but deducible. In the explicit case, they contain the binary value corresponding to the exact number of contents octets that follow. In the case where this number is unknown, they contain a reserved bit pattern indicating that the contents octets are delimited with a special EOC sequence. An EOC sequence consists of two octets whose binary values are zero.

Like the identifier octet(s), the length octet(s) may be expressed in more than one format: the short form, the long form, and the indefinite form. The use of a specific format is determined by three conditions: the form of the encoding (primitive or constructed), whether the number of contents octets is known in advance and the magnitude of this number. Primitive encodings are restricted from using the indefinite form since this would preclude the appearance of two consecutive "zero octets" within their contents octets. However in the case of a constructed type, the EOC octets always fall at the point in the encoding where the next identifier octet would be. Since zero is not a valid identifier octet (the UNIVERSAL 0 tag does not exist) there is no ambiguity. The choice of which format to use with constructed encodings is left to the user's discretion. Figure 26 illustrates the layout of these three forms.

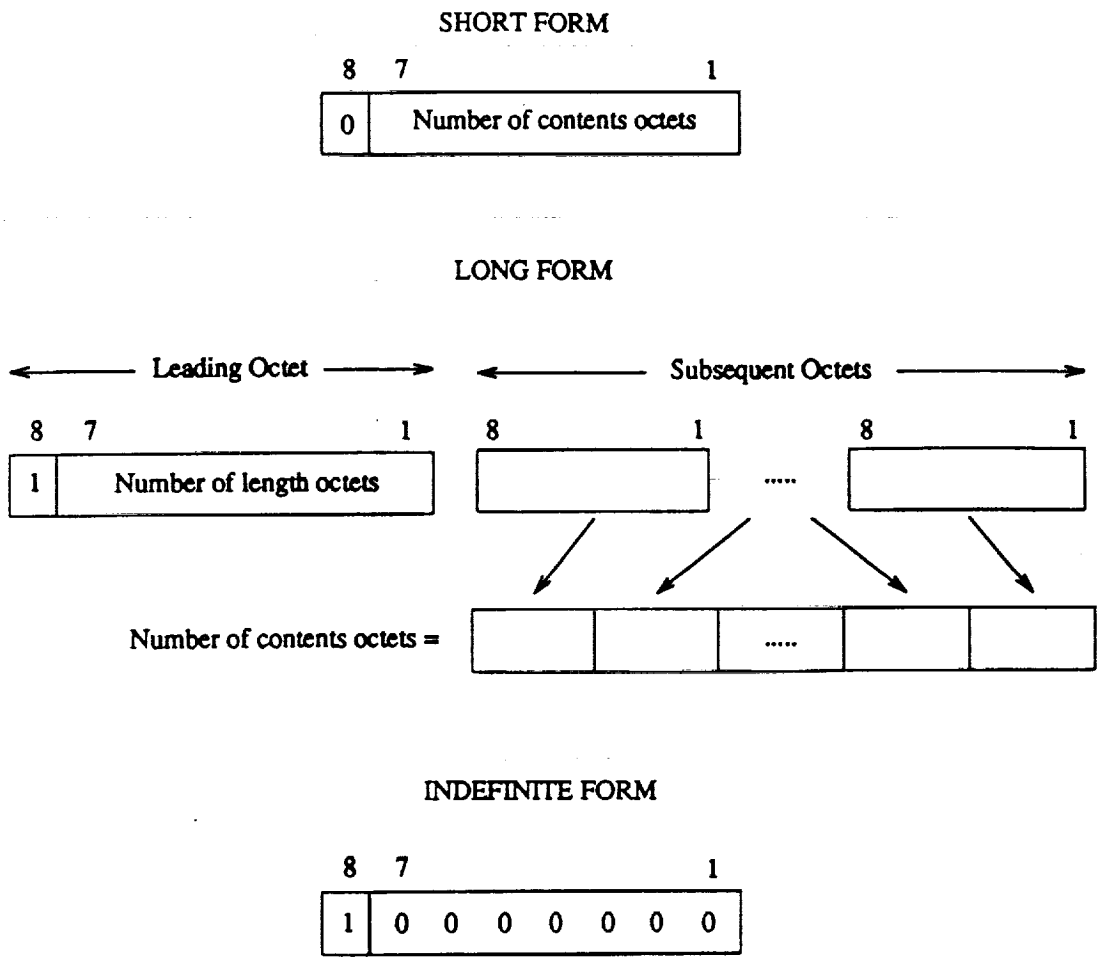



**Figure 5.** Length Octet Forms: short, long and indefinite

In the short form, the number of octets that the length information occupies is one. Bit 8 is required to be zero, and bits 7-1 contain the number of contents octets. Obviously, this form may be used only when the number of contents octets is less than or equal to 127. In the long form, the number of octets that the length information occupies can be anywhere from 2 to 127. In the leading octet, bit 8 is always set to 1, and bits 7-1 indicate the number of subsequent length octets required. Since the bit pattern 11111111 has been reserved for possible future extension, the maximum number of subsequent octets is 126. As illustrated in Figure 26, these subsequent octets are concatenated to form the binary value indicating the number of contents octets used in the encoding. Finally, the indefinite form requires only one octet containing the reserved bit

pattern indicated above. In relation to the other formats, this bit pattern represents the long form with zero subsequent length octets. From a logical perspective, this makes sense. No need for subsequent length octets exists since the contents octets are delimited by an EOC sequence.

### 3.0.3 The Contents Octet(s)

In a primitive encoding, the data value is directly represented by the contents octets. This may not, and in many instances will not, be exactly the same as the original native representation, depending on the host environment, the data type, and the magnitude of the value itself. For example, ISO 8825 specifies that negative integers are always in two's complement form and that the character values of an IA5String are in ASCII.[6] In a constructed encoding, the contents octets do not contain a direct representation of the data value, but instead contain further encodings. These nested encodings can in turn be constructed if necessary.

## 3.1 A BER Example

# 4. Building an ASN.1 Compiler

The purpose of an ASN.1 compiler is to automate the generation of logic required to encode and decode PDUs according to the transfer syntax, in this case the transfer syntax is BER. The relative benefit of an ASN.1 compiler is directly proportional to the stability of the protocol at hand. For mature protocols such as VT and FTAM, the likelihood of significant modifications is very small. Therefore, the need for an ASN.1 compiler is not a compelling issue. However, in an environment where a protocol is evolving and changing, an ASN.1 compiler offers tremendous

6. Many other characteristics of ISO 8825 also influence this change in representation. For example, integers are always encoded in the minimal number of octets, etc.

benefit.

Figure 2 depicts how an ASN.1 compiler might be used within an implementation. The source code processed by the compiler is the actual ASN.1 description of the data types of an Application Layer protocol. The object code consists of a set of encode and decode routines (*Wmp.c*[7] ), a run-time library (*runtime*), and an "include" file containing the corresponding APDU programming language declarations (*Wmp.h*). The encode, decode and run-time routines are subsequently embedded within the Presentation Layer implementation. The APDU declarations file is referenced within both the Application and the Presentation Layers.

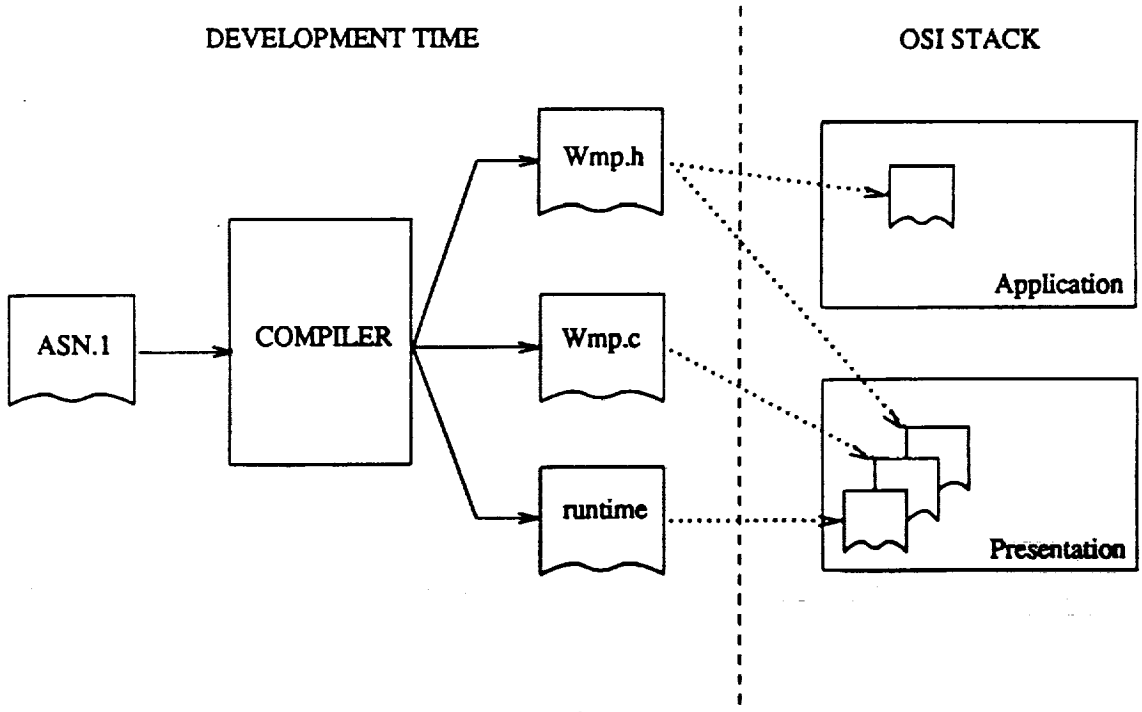



**Figure 6. Using an ASN.1 Compiler**

7. The names of the files generated by the compiler will not always be *Wmp.c* and *Wmp.h*. The names of these files always coincide with the name of the ASN.1 module.

## 4.1 Physical Specifications

The compiler is written in the C programming language and is approximately 5000 lines in length. Its front end was developed using the Unix utilities *Lex* and *Yacc*. The compiler accepts an ASN.1 module definition as input and generates two C files as output: an include file containing the C declarations of the APDU's, and a code file containing the two C functions *encode* and *decode*. A run-time library also provides a melange of general purpose routines. Unfortunately, the compiler does not support the entire ASN.1 standard; the macro notation, character sets other than IA5Strings, useful types, and external references are not presently implemented.

## 4.2 Generating Declarations

To create a transfer syntax, the ASN.1 compiler depends upon a precise knowledge of the APDU representations. It is therefore appropriate that the compiler assumes the responsibility of creating these declarations on behalf of the Application Layer. This provides a more robust implementation since it ensures that the ASN.1 data type definitions will be always consistent with the actual APDU representations. Clearly this is desirable considering that the relative benefit provided by an ASN.1 compiler hinges upon its ability to minimize the effects that modifications to the abstract syntax may cause. The reader's understanding of this mapping from the ASN.1 definitions to the APDU declarations is most important; it represents the first bridge between an ASN.1 definition and a working implementation.

Assuming that the reader is familiar with the C programming language, the clearest means of describing this aspect of the compiler is through an example. Let us examine the APDU declarations generated from the module definition of WMP that appears in Figure 1. These are shown in Figure 4.

```
typedef struct {
   int no_of_elems;
   char **elems;
} Participants_apdu;

typedef char *UTCTime_apdu;

typedef struct {
   Participants_apdu _unnamed_1;
   int id_number;
   struct {
      enum {time_of_completion,
         reason_for_failure} which_choice;
      union {
         UTCTime_apdu time_of_completion;
         char *reason_for_failure;
      } choice;
   } _unnamed_2;
} Result_apdu;

typedef struct {
   Participants_apdu *assignedto;
   UTCTime_apdu start_time;
   int id_number;
   int importance;
   char *description;
} Request_apdu;

typedef struct {
   enum {UTCTime,
      Participants,
      Result,
      Request} which_apdu;
   union {
      UTCTime_apdu UTCTime;
      Participants_apdu Participants;
      Result_apdu Result;
      Request_apdu Request;
   } apdu;
} Wmp_apdu;

typedef struct {
   int length;
   byte *value;
} Wmp_encoding;
```

**Figure 7. Declarations of WMP APDU's**

Although the above declarations do not represent the entire contents of the declarations file, this is sufficient to illustrate the important facts.

Each type assignment within an ASN.1 module definition produces a corresponding C typedef declaration. The *Request* data type, for instance, is represented by the typedef structure *Request_apdu*. As one would expect, it contains three members, each corresponding to one of the three elements within the original ASN.1 data type definition. Typedef declarations for the data types *Date* and *Result* also appear. All of these typedef declarations are subsequently referenced as members of a union that is embedded within the *Wmp_apdu* structure. *Wmp_apdu* is a variant record that represents any APDU of our WIMP protocol. Each of its variants corresponds to one of the data types defined within the original ASN.1 module. The *which_apdu* member identifies the type that is currently being represented by the union.

There are two subtle but important aspects of the above declarations that should be noted. First, the order of these typedef declarations does not correspond to the original ASN.1 definition. This is because forward references, while permitted within ASN.1, are not permitted within C. Consequently, the declaration of *Date_apdu* must precede the declaration of *Request_apdu*. Second, notice that the name of the second member of the *Request_apdu* structure is *_unnamed_1*. Since the second element of the original ASN.1 definition of the *Request* data type was not named, the corresponding C name must be supplied by the compiler itself. This illustrates why the use of named types within an ASN.1 element list enables the compiler to generate more readable C code.

## 4.3 Generating Code

The compiler generates two routines which are to be subsequently integrated into the Presentation Layer. Since actual C code of each is very detailed it will not be presented. The

reader may refer to Appendix A where a complete listing of the code generated for this particular example is given. Within this section the logical operation of each function is discussed in a very general sense. The corresponding function declarations are given below.

```
encode_Wmp (decoding, encoding)
        Wmp_apdu *decoding;
        Wmp_encoding *encoding;

decode_Wmp (encoding, decoding)
        byte *encoding;
        Wmp_apdu *decoding;
```

**Figure 8.** The Encoding and Decoding Functions

### 4.3.1 Encoding

The encoding function accepts the *decoding* argument as input and produces the *encoding* argument as output. The encoding process occurs in two phases. The purpose of the first phases is to calculate all of the length octet values of the encoding to be generated and to allocate an encoding buffer of the proper size. This task is performed by examining the contents of each value within the given APDU and making the appropriate run-time calls. These computed length values are inserted into a table which already contains the type octet values for each type that has been defined in the ASN.1 module. These values are subsequently extracted from this table in the later phase as the actual encoding is generated. The fact that the inner length octets of an ASN.1 encoding determine the values of the outer length octets explains why this length calculation phase is necessary. In other words, it is not possible to build an encoding from the inside out since the starting offset within the encoding buffer is unknown. The second phase focuses upon building the actual encoding. The type and length values from the previously built table are inserted into the encoding buffer and the actual values of the APDU are passed to the appropriate run-time routines which insert the contents octet values.

### 4.3.2 Decoding

The decoding function accepts the *encoding* argument as an input and produces the *decoding* argument as output. The decoding process is controlled by a finite state machine, implemented as a state table. Each entry of the state table contains an expected type octet value and a set of transitions. The finite state table for the Wmp module of Figure 10 is given below:

```
struct decode_state {
    struct {
        class c;
        form f;
        int id;
    } type;
    int match_state;
    int diff_state;
    int end_state;
    boolean matched
} Wmp_fsm[] = {
    {{universal, primitive, 22}, -1, 1, 0, false},
    {{universal, primitive, 2}, -1, 2, 0, false},
    {{universal, constructed, 16}, 3, -2, 0, false},
    {{universal, primitive, 22}, 4, -2, 0, false},
    {{universal, primitive, 22}, 5, -2, 0, false},
    {{universal, primitive, 22}, -1, -2, 0, false}
};
```

**Figure 9.** The Decoding Finite State Machine

The *match_state* element represents the transition that is to be made when the current type octet matches the type octet of the given state. As one might guess, the *diff_state* element represents the transition that is to be made when these two type octets differ. The *end_state* is necessary when the value being decoded requires that a set of states be executed an undeterminable number of times; this is the case when the value is a set type, a set-of type, a sequence-of type, or a recursive type.

The actual decoding process consists of a large switch statement that is iteratively executed until either the decoding is successfully completed, or the decoding fails.

## 4.4 The Run-Time Library

Most of the "bit-level" details of the encoding and decoding process are contained in the run-time library. This library consists of a set of C routines that perform very specific functions. The nature of these functions does not vary with respect to the ASN.1 definition; that is, the work performed by these run-time routines is applicable to the encoding and decoding of almost all ASN.1 definitions. An overview of these run-time routines is given below.

The ASN.1 compiler's run-time library consists of the following routines:

1. *Dec_typlen* : This function decodes the type and length octets of an an encoding. It supports both the five bit form as well as the extended form of the type octet(s) as well as the short, long, and indefinite forms of length octet(s).

2. *Enc_typlen* : This function decodes the type and length octets of an encoding. It supports both the five bit form as well as the extended form of the type octet(s) as well as the short, long, and indefinite forms of length octet(s).

3. *Len_len* : This function determines the length of a length octet value. In other words, it determines how many length octets are required to hold a given length value. This routine is called by the encoding routine when it is computing the length of the encoding.

4. *Int_len* : This function determines the number of contents octets required to encode a given integer value.

5. *Enc_int* : This function inserts the encoding of an integer value into the contents octet(s) of an encoding. In accordance with clause 8.2 of ISO 8825, the number of contents octets produced is always the minimum required to represent the integer value.

6. *Dec_int* : This function returns the integer value represented by the contents octet(s) of an encoding.

7. *Enc_bool* : This function inserts the encoding of a boolean value into the contents octet(s) of an encoding.

8. *Dec_bool* : This function returns the boolean value represented by the contents octet(s) of an encoding. A boolean value in C is declared as an unsigned char.

9. *Enc_ia5str* : This function inserts the encoding of a IA5String value into the contents octet(s) of an encoding.

10. *Dec_ia5str* : This function returns the IA5String value represented by the contents octet(s) of an encoding.

11. *Enc_bstr* : This function inserts the encoding of a BIT STRING value into the contents octet(s) of an encoding.

12. *Dec_bstr* : This function returns the BIT STRING value represented by the contents octet(s) of an encoding.

13. *Enc_ostr* : This function inserts the encoding of a OCTET STRING value into the contents octet(s) of an encoding.

14. *Dec_ostr* : This function returns the OCTET STRING value represented by the contents octet(s) of an encoding.

15. *Obj_id_len* : This function calculates the number of contents octets required to encode a given OBJECT IDENTIFIER value.

16. *Enc_ostr* : This function inserts the encoding of a OBJECT IDENTIFIER value into the contents octet(s) of an encoding.

17. *Dec_ostr* : This function returns the OBJECT IDENTIFIER value represented by the contents octet(s) of an encoding.

## 4.5 Performance

As a more substantial test, the F-INITIALIZE-request PDU of the FTAM Application protocol[3] was given to the ASN.1 compiler. With the exception of converting two Graphic String data types to IA5Strings and modifying one tag,[8] the FTAM definition used to conduct this performance measurement should represent the overhead one would expect to encounter in reality. Although the other PDUs of FTAM were reduced to integer data types, their explicit tags were left intact. Since these types correspond to transitions within the decoding finite state machine that are never be made, the reduction of these types is irrelevant to the measured performance.

Given the definition described above, the ASN.1 compiler generated a declaration file of 194 lines and a C code file of 2849 lines. Running on a Sun 3/140 workstation, the time to encode the F-INITIALIZE-request PDU was approximately 5.11 milliseconds, and the time to decode was approximately 4.97 milliseconds.[9]

# 5. Conclusions

Through the process of building the compiler many insights into the ASN.1 and BER were obtained. This section presents these observations in an effort to illuminate the answer to the question of how well ISO 8824 and ISO 8825 solve the problem of network heterogeneity. When approaching this issue it is essential that one bear in mind the kinds of characteristics that determine the context that makes such an evaluation possible: efficiency, clarity, applicability,

8. The explicit tag of the Protocol-Version data type was changed from [0] to [9]. This was necessary due to an ambiguity within FTAM.

9. Timings were obtained using the Unix *time* command whereupon the reported user and system times were combined. The Sun workstation was dedicated to a single user when these measurements were taken.

and ease of implementation.

## 5.1 Ambiguity

One of the more surprising discoveries that resulted from our experience with the ASN.1 standard is the fact that the concept of ambiguity is not well defined. In the introduction of ISO 8824 a footnote indirectly addresses this issue in the context of tagging. It states:

> Encoding rules always carry the tag of the type, explicitly or implicitly, with any representation of a value of the type. The restrictions placed on the use of the notation are designed to ensure that the tag is sufficient to unambiguously determine the actual type, provided the applicable type definitions are available.

The reference to *the* tag (singular) in the above statement suggests that if ever a specific tag value indicates more than one type in any given context, then the ASN.1 definition is ambiguous. If this is indeed the correct interpretation, then the logic required to decode an encoding is LR(0).[10] This coincides with the finite state machine approach that is used to perform the decoding within our ASN.1 compiler. Yet, this same interpretation of ambiguity also suggests that the official ASN.1 definition of FTAM is ambiguous!

Looking at the partial definition of FTAM that appears in Appendix B, notice that the data type *PDU* is a choice consisting of three untagged alternatives. Since these elements do not possess explicit tags, an encoding of this data type is identical to the encoding of the selected alternative. Furthermore, notice that the *FTAM-Regime-PDU* alternative of the *PDU* data type is also a choice whose first element, *F-INITIALIZE-request*, is assigned the tag context 0. This means that the first tag of an encoding of the *F-INITIALIZE-request* PDU will always be context 0. Now note that the *Protocol-Version* data type that appears later in the module also possesses

10. Other ASN.1 compilers refer to decoding as "parsing".

the explicit tag context 0. Since the definition of this type appears at the outermost lexical level, as does the definition of *PDU*, both of these data types represent potential PDUs. Consequently, when the generated FTAM decoding routine encounters a context 0 tag as the first type octet of an encoding, it has no way of knowing whether or not the value being represented is a *Protocol-Version* PDU or a *F-INITIALIZE-request* PDU. In this particular case, a two tag lookahead is necessary to disambiguate the encoding.

From the name *PDU* it is natural to assume that this data type is the root of the FTAM definitions and that all FTAM PDU's originate from this type. This heuristic could therefore be used to resolve this particular ambiguity. However, this is clearly an assumption since there is no construct of ASN.1 being used to indicate that this condition holds true. In as much as forward references are allowed in the notation, it is not possible to make this assumption based on the mere fact that this definition appears first. Consequently, all data types that appear at the outermost lexical level of a module must be considered PDUs in their entirety and the ambiguity remains. In fact, since the elements of the *PDU* data type do not possess explicit tags, the absence or presence of this type definition does not influence the formulated encodings in any way; in other words, this data type is extraneous with respect to how an FTAM encoding is built and interpreted.

It is true that the FTAM standard does not intend that the *Protocol-Version* data type be considered a PDU by itself. In reality it is a supporting type definition that is referenced as a named type within other PDU definitions. Yet, the compiler has no means of making this distinction based on the notation itself.[11] Therefore, every type definition that is not lexically

11. Using the heuristic that any data type mentioned within another type is necessarily a supporting type would preclude recursive definitions.

embedded within another type definition must be treated as a PDU. As a general approach to the issue of ambiguity, requiring more than a single tag lookahead would require any ASN.1 compiler to generate an LR($k$) implementation. This would almost certainly prevent the realization of efficient implementations.

The ASN.1 definition of what constitutes an ambiguous instance of the notation requires clarification.

## 5.2 Library Management

The unrestricted manner in which type or value definitions within other ASN.1 modules may be referenced presents another area of the ASN.1 standard that appears to be inadequately defined. When an ASN.1 user wishes to reference a type or value that has been defined within another module the External Reference construct may be used. As explained in section 3.4.3, an external reference consists of the specification of the module name where the definition appears and the actual name of the desired type or value. A proposal to enable the ASN.1 user to exert explicit control over the importing and exporting of definitions is currently under consideration in the pending draft addendum. However, a potential problem which has been overlooked by the definition of this external referencing capability involves the potential for mutually dependent module definitions. Consider the following modules:

```
Module-A DEFINITIONS ::= BEGIN
   Type-A1 ::= BIT STRING
   Type-A2 ::= SEQUENCE {
      Module-B.Type-B1
   }
END

Module-B DEFINITIONS ::= BEGIN
   Type-B1 ::= OCTET STRING
   Type-B2 ::= SEQUENCE {
      Module-A.Type-A1
   }
END
```

**Figure 10.** Mutually Dependent Modules

In this example, two ASN.1 modules are mutually dependent since each references a data type that is defined within the other.[12] This potential for mutual dependencies requires that an ASN.1 compiler generate partially complete intermediate code files and makes the automated processing of these definitions extremely difficult. This problem is attributable to the absence of a well defined library management concept within ISO 8824.

### 5.2.1 Linear Elaboration

The problem of mutually dependent modules can be solved if ISO 8824 were to establish a requirement of linear elaboration.[4] The Ada programming language effectively utilizes this concept to prevent the possibility of mutual dependencies among compilation units. *Elaboration* is defined as the process by which an entity is brought into existence. According to the definition of Ada, this marks the point at which the name of a declarative item is bound to its type. The name of an Ada entity may not be used before the elaboration of the declarative item that declares this entity takes place. With respect to compilation units, the concept of linear elaboration

12. Mutually dependent modules appear in the X.500 series of standards.

requires that a noncircular ordering of all compilation units referenced, either directly or indirectly, by a given program must exist. If no such ordering can be found, the program is illegal.

This same requirement could be applied to the ASN.1 standard to prevent the possibility of mutual dependencies.

### 5.2.2 Useful Types

A set of predefined data types, called useful types, is specified in section three of ISO 8824. Four data types are defined: Generalized time, Universal time, the External type, and the Object Descriptor type. One problem associated with the manner in which these types are defined concerns their disposition relative to the rest of the standard. Specifically, the means through which their names are made visible to all module definitions is not clear. Hence, their relationship to the ASN.1 standard itself is unclear. The development of a library concept would also serve to solidify the disposition of these types.

Like Ada's package STANDARD, a predefined module of these useful types should be defined. This would consequently allow any ASN.1 module to explicitly import these types whenever their use is required. Alternatively, this importing of these types could be defined as implicit as is the case with package STANDARD. In any regard, the disposition of these types as a predefined environment would be clarified and therefore strengthened if a module were to be formally defined.

## 5.3 Efficiency versus Generality

The development of any standard seems to entail a perpetual series of compromises. Issues concerning generality and efficiency are typically in conflict and a balance of the two is often the

best that can be achieved. For the most part, ASN.1 and BER adequately strikes this balance. However, there are particular instances where the issue of generality has been favored over considerations of efficiency to a questionable degree.

### 5.3.1 Encoding Integers

As stated in section 3.3.6.1, ISO 8825 requires that all integer values be encoding in a minimum number of octets. This requirement prohibits integer encodings from being unnecessarily large and prevents the pathological case where an integer value may be transferred with an arbitrarily number of leading octets that contain either all zeros or all ones. Nonetheless, the cost of this decision is expensive with respect to efficiency. It has been shown that the encoding and decoding of an ASN.1 integer value, in contrast to a simple memory copying technique, decreases the transfer rate by a factor that ranges from 5 to 20 depending upon the host system.[5] Moreover, in this same set of experiments it was shown that a fixed length approach reduced the encoding time by a factor of 5-6.

To support the requirement of integer encoding minimality the run-time system must perform a series of checks to determine the size of each integer value. This can be accomplished through successive range checks or successive logical operations on the leading byte of the value. However, regardless of which approach is used, the process is unnecessarily slow. Requiring integers to be encoded in a minimum number of bytes is ill-advised with respect to performance.

### 5.3.2 Object Identifiers

The encoding and decoding of object identifier values is an unnecessarily inefficient process. Comprised of a series of numeric values, called component values, an object identifier is encoded by representing each component value as a series of seven-bit quantities. These seven-bit quantities are concatenated to form each component. The leading bit of each contents octet is

used to demarcate the boundaries of these component values. To further contribute to this inefficiency, the first two components of any object are handled as a special case. Since the value of the first component is restricted to the range 0..2, the encoding of the first two components may be represented using the equation: $ 40(X) ¯ + ¯ Y $ where X denotes the value of the first component and Y denotes the value of the second. This requires that the runtime routine that performs this encoding and decoding must use its knowledge of the object identifier tree structure to determine the values of X and Y. The application of this knowledge is expensive.

The use of seven-bit quantities is not desirable since bit masking or arithmetic shifting is always necessary to isolate the value. Furthermore, it is not clear whether a savings of one octet warrants the computationally expensive combination of the first two component values.

## 5.4 Explicit Tagging

From a philosophical perspective the concept of explicit tagging seems erroneous; it thrusts the responsibility of preventing ambiguity onto the user. It is certainly possible to establish a standard, canonical ordering of the type definitions within any module and to determine tag values from this ordering. Consequently, a compiler could potentially generate these tags and assume this responsibility instead of the user. As long as the generated tags are predictable, economical, and unique, the users needs would be met. For example, a simple top down, left to right sequential numbering scheme would fulfill these three requirements.

If such a sequential tagging generation algorithm were adopted, tag values would not be reusable in the case of mutually exclusive contexts. Therefore, the point at which extended tag values are necessary might arrive sooner than before and would cause the average length of an encoding to increase. However, this point would depend on the extent of the mutually exclusive contexts: note that the tag class values would no longer be required under this new scheme and,

therefore, the capacity of the tag id field could be extended from 0..30 to 0..126.

Clearly it is more desirable to have these tags generated in an automated manner since the issue of ambiguity can be subtle and error-prone.

## 5.5 Sets

The only difference between the ASN.1 sequence type and the ASN.1 set type is that the order of elements within a set is not considered significant. Therefore, opting to define a data type as a set rather than a sequence is less restrictive. There is an intuitive tendency to equate restrictiveness with inefficiency; after all, removing this restriction appears to allow the elements to be reordered during transfer if this presents itself as a more efficient alternative. However, this freedom is not more efficient. In fact the decoding of a set value is far less efficient than a sequence due to this unpredictable ordering and, ironically, it is extremely unlikely that the order that these elements are received will ever differ from the order that appears in the original definition.[13]

The elements of a set value are decoding by a cycle of states within the decoding finite state machine. This cycle is executed until it is determined that the end of the encoding has been reached. As each element is encountered a boolean flag within the decoding state table, *matched*, is checked and set to indicate that this decoding has occurred. After the appropriate runtime routine performs the actual decoding, a check is made to determine if the end of the encoding has been reached. If this check succeeds, then the *matched* flag of each element must be re-checked in the state table to ensure that all the element values which were not marked as optional or had

13. The Session Layer does not provide the capability of reordering portions of a single PDU. It is therefore the exclusive responsibility of the ASN.1 compiler to perform this reordering. Yet, it is doubtful that circumstances exist where this would be desirable.

not been assigned a default value have been represented within the encoding. If all of these checks are successful the decoding terminates successfully.

In contrast, the decoding of a sequence value is a far less time consuming activity. Since the order of elements is significant, decoding is simply a matter of checking each tag as it is encountered to ensure that it matches that which is expected. If this tag check ever fails, the decoding is terminated as a failure. Note that the end of the encoding does not require special processing and there is no need for the *matched* flag to be set or checked.

The inclusion of set types in ISO 8824 appears esoteric and predicated on the need for completeness. Their use within an actual implementation can be deceptive where efficiency is concerned.

## 5.6 Macros

The Macro Notation enables a user to alter the grammar of the ASN.1 language "on the fly". As such, the existence of macros makes developing a conformant implementation inordinately difficult. Although certain programming languages are capable of solving this class of problem,[14] these languages do not possess the efficiency to make their use advisable in a network implementation. Hence, the current definition of macros poses an obstacle.

Philosophically, the presence of a macro notation is inappropriate. As a descriptive mechanism that defines the informational content of an application protocol, one of ASN.1's most valuable benefits is the concise and accurate means of *human* communication that it represents. In this respect, it is critical that its form remain standard. This enables ASN.1 to

14. programming languages like Icon or LISP

effectively bridge the gap between the precise communication of computers and the imprecise communication of human beings.

With the use of macros, the ASN.1 language can assume an amorphic form that is determined by the personal tastes of the ASN.1 writer. This can only serve to reduce the overall quality of the notation itself.

## *REFERENCES*

1. ISO 8824: 1987(E), "Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)", International Organization for Standardization, Switzerland, December 15, 1987.

2. ISO 8825: 1987(E), "Information processing systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)", International Organization for Standardization, Switzerland, November 15, 1987.

3. ISO 8571/4: 1986(E), "Information processing systems - Open Systems Interconnection - File transfer, access and management - Part 4: The file protocol specification", International Organization for Standardization, Switzerland, August 7, 1986.

4. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1851A, American National Standards Institute, Inc., February, 17 1983.

5. Christian Huitema and Assem Doghri. "A High Speed Approach for the OSI Presentation Protocol", Proceedings of the *IFIP International Workshop on Protocols for High-Speed Networks*, Zurich, Switzerland, May 9, 1989.