Using the Open Source ASN.1 Compiler

Lev Walkin <vlm@lionet.info>

18th November 2005

This document describes asn1c-0.9.20 $Revision : 1.28$

Contents

Part I

Using the ASN.1 Compiler

Chapter 1

Introduction to the ASN.1 Compiler

The purpose of the ASN.1 compiler, of which this document is part, is to convert the specifications in ASN.1 notation into some other language. At this moment, only C and C++ target languages are supported, the latter is in upward compatibility mode.

The compiler reads the specification and emits a series of target language structures (C's structs, unions, enums) describing the corresponding ASN.1 types. The compiler also creates the code which allows automatic serialization and deserialization of these structures using several standardized encoding rules (BER, DER, XER).

For example, suppose the following ASN.1 module is given[1]:

```
RectangleTest DEFINITIONS ::=
BEGIN

Rectangle ::= SEQUENCE {
    height  INTEGER,        -- Height of the rectangle
    width   INTEGER         -- Width of the rectangle
}

END
```

The compiler would read this ASN.1 definition and produce the following C type[2]:

```
typedef struct Rectangle_s {
    int height;
    int width;
} Rectangle_t;
```

[1] Please look into Part II on page 27 for a quick reference on how to understand the ASN.1 notation.

[2] *-fnative-types* compiler option is used to produce basic C *int* types instead of infinite width INTEGER_t structures. See Section 1.3 on page 9.

It would also create the code for converting this structure into platform-independent wire representation (a serializer API) and the decoder of such wire representation back into local, machine-specific type (a deserializer API).

1.1 Quick start with asn1c

After building and installing the compiler, the *asn1c*[3] command may be used to compile the ASN.1 module[4]:

```
asn1c <module.asn1>
```

If several ASN.1 modules contain interdependencies, all of the files must be specified altogether:

```
asn1c <module1.asn1> <module2.asn1> ...
```

The compiler **-E** and **-EF** options are used for testing the parser and the semantic fixer, respectively. These options will instruct the compiler to dump out the parsed (and fixed, if **-F** is involved) ASN.1 specification as it was "understood" by the compiler. It might be useful to check whether a particular syntactic construction is properly supported by the compiler.

```
asn1c -EF <module-to-test.asn1>
```

The **-P** option is used to dump the compiled output on the screen instead of creating a bunch of .c and .h files on disk in the current directory. You would probably want to start with **-P** option instead of creating a mess in your current directory. Another option, **-R**, asks compiler to only generate the files which need to be generated, and supress linking in the numerous support files.

Print the compiled output instead of creating multiple source files:

```
asn1c -P <module-to-compile-and-print.asn1>
```

1.2 Recognizing compiler output

After compiling, the following entities will be created in your current directory:

- A set of .c and .h files, generally a single pair for each type defined in the ASN.1 specifications. These files will be named similarly to the ASN.1 types (*Rectangle.c* and *Rectangle.h* for the RectangleTest ASN.1 module defined in the beginning of this document).

[3] The 1 symbol in asn**1**c is a digit, not an ”ell” letter.

[4] This is probably **not** what you want to try out right now – read through the rest of this chapter and check the Section 1.3 on the facing page to find out about **-P** and **-R** options.

- A set of helper .c and .h files which contain generic encoders, decoders and other useful routines. There will be quite a few of them, some of them even are not always necessary, but the overall amount of code after compilation will be rather small anyway.
- A *Makefile.am.sample* file mentioning all the files created at the earlier steps. This file is suitable for either automake suite or the plain ‘make‘ utility.

It is your responsibility to create .c file with the *int main()* routine.

In other words, after compiling the Rectangle module, you have the following set of files: { Makefile.am.sample, Rectangle.c, Rectangle.h, **...** }, where **”...”** stands for the set of additional ”helper” files created by the compiler. If you add a simple file with the *int main()* routine, it would even be possible to compile everything with the single instruction:

```
cc -I. -o rectangle.exe *.c   # It could be that simple
```

Refer to the Chapter 3 on page 19 for a sample *int main()* routine.

1.3 Command line options

The following table summarizes the asn1c command line options.

Overall Options	**Description**
-E	Stop after the parsing stage and print the reconstructed ASN.1 specification code to the standard output.
-F	Used together with -E, instructs the compiler to stop after the ASN.1 syntax tree fixing stage and dump the reconstructed ASN.1 specification to the standard output.
-P	Dump the compiled output to the standard output instead of cre- ating the target language files on disk.
-R	Restrict the compiler to generate only the ASN.1 tables, omit- ting the usual support code.
-S <*directory*>	Use the specified directory with ASN.1 skeleton files.
-X	Generate the XML DTD for the specified ASN.1 modules.

Warning Options	**Description**
-Werror	Treat warnings as errors; abort if any warning is produced.
-Wdebug-lexer	Enable lexer debugging during the ASN.1 parsing stage.
-Wdebug-fixer	Enable ASN.1 syntax tree fixer debugging during the fixing stage.
-Wdebug-compiler	Enable debugging during the actual compile time.

Language Options	**Description**
-fall-defs-global	Normally the compiler hides the definitions (asn_DEF_xxx) of the inner structure elements (members of SEQUENCE, SET and other types). This option makes all such definitions global. Enabling this option may pollute the namespace by making lots of asn_DEF_xxx structures globally visible, but will allow you to manipulate (encode and decode) the individual members of any complex ASN.1 structure.
-fbless-SIZE	Allow SIZE() constraint for INTEGER, ENUMERATED, and other types for which this constraint is normally prohibited by the standard. This is a violation of an ASN.1 standard and compiler may fail to produce the meaningful code.
-fcompound-names	Use complex names for C structures. Using complex names prevents name clashes in case the module reuses the same identifiers in multiple contexts.
-findirect-choice	When generating code for a CHOICE type, compile the CHOICE members as indirect pointers instead of declaring them inline. Consider using this option together with **-fno-include-deps** to prevent circular references.
-fknown-extern-type=<*name*>	Pretend the specified type is known. The compiler will assume the target language source files for the given type have been provided manually.
-fnative-types	Use the native machine's data types (int, double) whenever possible, instead of the compound INTEGER_t, ENUMERATED_t and REAL_t types.
-fno-constraints	Do not generate ASN.1 subtype constraint checking code. This may produce a shorter executable.
-fno-include-deps	Do not generate courtesy #include lines for non-critical dependencies.
-funnamed-unions	Enable unnamed unions in the definitions of target language's structures.
-fskeletons-copy	Copy support files rather than symlink them.

Output Options	**Description**
-print-constraints	When -EF are also specified, this option forces the compiler to explain its internal understanding of subtype constraints.
-print-lines	Generate "– #line" comments in -E output.

Chapter 2

Using the ASN.1 Compiler

2.1 Invoking the ASN.1 helper code

First of all, you should include one or more header files into your application. Typically, it is enough to include the header file of the main PDU type. For our Rectangle module, including the Rectangle.h file is sufficient:

```
#include <Rectangle.h>
```

The header files defines the C structure corresponding to the ASN.1 definition of a rectangle and the declaration of the ASN.1 type descriptor, which is used as an argument to most of the functions provided by the ASN.1 module. For example, here is the code which frees the Rectangle_t structure:

```
Rectangle_t *rect = ...;

asn_DEF_Rectangle.free_struct(&asn_DEF_Rectangle,
    rect, 0);
```

This code defines a *rect* pointer which points to the Rectangle_t structure which needs to be freed. The second line invokes the generic *free_struct()* routine created specifically for this Rectangle_t structure. The *asn_DEF_Rectangle* is the type descriptor, which holds a collection of routines to deal with the Rectangle_t structure.

The following member functions of the asn_DEF_Rectangle type descriptor are of interest:

ber_decoder This is the generic *restartable*[1] BER decoder (Basic Encoding Rules). This decoder would create and/or fill the target structure for you. Please refer to Section 2.1.1 on the following page.

[1] Restartable means that if the decoder encounters the end of the buffer, it will fail, but may later be invoked again with the rest of the buffer to continue decoding.

der_encoder This is the generic DER encoder (Distinguished Encoding Rules). This encoder will take the target structure and encode it into a series of bytes. Please refer to Section 2.1.2 on page 14.

xer_encoder This is the XER encoder (XML Encoding Rules). This encoder will take the target structure and represent it as an XML (text) document using either BASIC-XER or CANONICAL-XER encoding rules. Please refer to Section 2.1.3 on page 15.

xer_decoder This is the generic XER decoder. It takes both BASIC-XER or CANONICAL-XER encodings and deserializes the data into a local, machine-dependent representation. Please refer to Section 2.1.4 on page 16.

check_constraints Check that the contents of the target structure are semantically valid and constrained to appropriate implicit or explicit subtype constraints. Please refer to Section 2.1.5 on page 16.

print_struct This function convert the contents of the passed target structure into human readable form. This form is not formal and cannot be converted back into the structure, but it may turn out to be useful for debugging or quick-n-dirty printing. Please refer to Section 2.1.6 on page 17.

free_struct This is a generic disposal which frees the target structure. Please refer to Section 2.1.7 on page 17.

Each of the above function takes the type descriptor (*asn_DEF_...*) and the target structure (*rect*, in the above example).

2.1.1 Decoding BER

The Basic Encoding Rules describe the most widely used (by the ASN.1 community) way to encode and decode a given structure in a machine-independent way. Several other encoding rules (CER, DER) define a more restrictive versions of BER, so the generic BER parser is also capable of decoding the data encoded by CER and DER encoders. The opposite is not true.

The ASN.1 compiler provides the generic BER decoder which is implicitly capable of decoding BER, CER and DER encoded data.

The decoder is restartable (stream-oriented), which means that in case the buffer has less data than it is expected, the decoder will process whatever there is available and ask for more data to be provided. Please note that the decoder may actually process less data than it was given in the buffer, which means that you must be able to make the next buffer contain the unprocessed part of the previous buffer.

Suppose, you have two buffers of encoded data: 100 bytes and 200 bytes.

- You may concatenate these buffers and feed the BER decoder with 300 bytes of data, or

- You may feed it the first buffer of 100 bytes of data, realize that the ber_decoder consumed only 95 bytes from it and later feed the decoder with 205 bytes buffer which consists of 5 unprocessed bytes from the first buffer and the additional 200 bytes from the second buffer.

This is not as convenient as it could be (like, the BER encoder could consume the whole 100 bytes and keep these 5 bytes in some temporary storage), but in case of existing stream based processing it might actually fit well into existing algorithm. Suggestions are welcome.

Here is the simplest example of BER decoding.

```
Rectangle_t *
simple_deserializer(const void *buffer, size_t buf_size) {
    Rectangle_t *rect = 0;    /* Note this 0! */
    asn_dec_rval_t rval;

    rval = asn_DEF_Rectangle.ber_decoder(0,
          &asn_DEF_Rectangle,
          (void **)&rect,
          buffer, buf_size,
          0);

    if(rval.code == RC_OK) {
        return rect;          /* Decoding succeeded */
    } else {
        /* Free partially decoded rect */
        asn_DEF_Rectangle.free_struct(
            &asn_DEF_Rectangle, rect, 0);
        return 0;
    }
}
```

The code above defines a function, *simple_deserializer*, which takes a buffer and its length and is expected to return a pointer to the Rectangle_t structure. Inside, it tries to convert the bytes passed into the target structure (rect) using the BER decoder and returns the rect pointer afterwards. If the structure cannot be deserialized, it frees the memory which might be left allocated by the unfinished *ber_decoder* routine and returns 0 (no data). (This **freeing is necessary** because the ber_decoder is a restartable procedure, and may fail just because there is more data needs to be provided before decoding could be finalized). The code above obviously does not take into account the way the *ber_decoder()* failed, so the freeing is necessary because the part of the buffer may already be decoded into the structure by the time something goes wrong.

A little less wordy would be to invoke a globally available *ber_decode()* function instead of dereferencing the asn_DEF_Rectangle type descriptor:

```
rval = ber_decode(0, &asn_DEF_Rectangle, (void **)&rect,
    buffer, buf_size);
```

Note that the initial (asn_DEF_Rectangle.ber_decoder) reference is gone, and also the last argument (0) is no longer necessary.

These two ways of BER decoder invocations are fully equivalent.

The BER de*coder* may fail because of (*the following RC_… codes are defined in ber_decoder.h*):

- RC_WMORE: There is more data expected than it is provided (stream mode continuation feature);
- RC_FAIL: General failure to decode the buffer;
- … other codes may be defined as well.

Together with the return code (.code) the asn_dec_rval_t type contains the number of bytes which is consumed from the buffer. In the previous hypothetical example of two buffers (of 100 and 200 bytes), the first call to ber_decode() would return with .code = RC_WMORE and .consumed = 95. The .consumed field of the BER decoder return value is **always** valid, even if the decoder succeeds or fails with any other return code.

Please look into ber_decoder.h for the precise definition of ber_decode() and related types.

2.1.2 Encoding DER

The Distinguished Encoding Rules is the *canonical* variant of BER encoding rules. The DER is best suited to encode the structures where all the lengths are known beforehand. This is probably exactly how you want to encode: either after a BER decoding or after a manual fill-up, the target structure contains the data which size is implicitly known before encoding. Among other uses, the DER encoding is used to encode X.509 certificates.

As with BER decoder, the DER encoder may be invoked either directly from the ASN.1 type descriptor (asn_DEF_Rectangle) or from the stand-alone function, which is somewhat simpler:

```
/*
 * This is the serializer itself,
 * it supplies the DER encoder with the
 * pointer to the custom output function.
 */
ssize_t
simple_serializer(FILE *ostream, Rectangle_t *rect) {
    asn_enc_rval_t er;  /* Encoder return value */

    er = der_encode(&asn_DEF_Rect, rect,
        write_stream, ostream);
    if(er.encoded == -1) {
        /*
```

```
         * Failed to encode the rectangle data.
         */
        fprintf(stderr, ”Cannot encode %s: %s\n”,
            er.failed_type->name,
            strerror(errno));
        return -1;
    } else {
        /* Return the number of bytes */
        return er.encoded;
    }
}
```

As you see, the DER encoder does not write into some sort of buffer or something. It just invokes the custom function (possible, multiple times) which would save the data into appropriate storage. The optional argument *app_key* is opaque for the DER encoder code and just used by *_write_stream()* as the pointer to the appropriate output stream to be used.

If the custom write function is not given (passed as 0), then the DER encoder will essentially do the same thing (i.e., encode the data) but no callbacks will be invoked (so the data goes nowhere). It may prove useful to determine the size of the structure's encoding before actually doing the encoding[2].

Please look into der_encoder.h for the precise definition of der_encode() and related types.

2.1.3 Encoding XER

The XER stands for XML Encoding Rules, where XML, in turn, is eXtensible Markup Language, a text-based format for information exchange. The encoder routine API comes in two flavors: stdio-based and callback-based. With the callback-based encoder, the encoding process is very similar to the DER one, described in Section 2.1.2 on the preceding page. The following example uses the definition of write_stream() from up there.

```
/*
 * This procedure generates the XML document
 * by invoking the XER encoder.
 * NOTE: Do not copy this code verbatim!
 *       If the stdio output is necessary,
 *       use the xer_fprint() procedure instead.
 *       See Section 2.1.6 on page 17.
 */
int
print_as_XML(FILE *ostream, Rectangle_t *rect) {
    asn_enc_rval_t er;  /* Encoder return value */
```

[2] It is actually faster too: the encoder might skip over some computations which aren't important for the size determination.

```
        er = xer_encode(&asn_DEF_Rectangle, rect,
            XER_F_BASIC, /* BASIC-XER or CANONICAL-XER */
            write_stream, ostream);

        return (er.encoded == -1) ? -1 : 0;
    }
```

Please look into xer_encoder.h for the precise definition of xer_encode() and related types.

See Section 2.1.6 for the example of stdio-based XML encoder and other pretty-printing suggestions.

2.1.4 Decoding XER

The data encoded using the XER rules can be subsequently decoded using the xer_decode() API call:

```
    Rectangle_t *
    XML_to_Rectangle(const void *buffer, size_t buf_size) {
        Rectangle_t *rect = 0; /* Note this 0! */
        asn_dec_rval_t rval;

        rval = xer_decode(0, &asn_DEF_Rectangle, (void **)&rect,
            buffer, buf_size);
        if(rval.code == RC_OK) {
            return rect;          /* Decoding succeeded */
        } else {
            /* Free partially decoded rect */
            asn_DEF_Rectangle.free_struct(
                &asn_DEF_Rectangle, rect, 0);
            return 0;
        }
    }
```

The decoder takes both BASIC-XER and CANONICAL-XER encodings.

The decoder shares its data consumption properties with BER decoder; please read the Section 2.1.1 on page 12 to know more.

Please look into xer_decoder.h for the precise definition of xer_decode() and related types.

2.1.5 Validating the target structure

Sometimes the target structure needs to be validated. For example, if the structure was created by the application (as opposed to being decoded from some external source), some important information required by the ASN.1 specification might be missing. On the other hand, the successful decoding of the data from some external source does not necessarily mean that the data is fully valid either. It might well be the case that the specification describes some subtype constraints that were not taken into account during decoding, and it would actually be useful to perform the last check when the data is ready to be encoded or when the data has just been decoded to ensure its validity according to some stricter rules.

The asn_check_constraints() function checks the type for various implicit and explicit constraints. It is recommended to use asn_check_constraints() function after each decoding and before each encoding.

Please look into constraints.h for the precise definition of asn_check_constraints() and related types.

2.1.6 Printing the target structure

There are two ways to print the target structure: either invoke the print_struct member of the ASN.1 type descriptor, or using the asn_fprint() function, which is a simpler wrapper of the former:

```
asn_fprint(stdout, &asn_DEF_Rectangle, rect);
```

Please look into constr_TYPE.h for the precise definition of asn_fprint() and related types.

Another practical alternative to this custom format printing would be to invoke XER encoder. The default BASIC-XER encoder performs reasonable formatting for the output to be useful and human readable. To invoke the XER decoder in a manner similar to asn_fprint(), use the xer_fprint() call:

```
xer_fprint(stdout, &asn_DEF_Rectangle, rect);
```

See Section 2.1.3 on page 15 for XML-related details.

2.1.7 Freeing the target structure

Freeing the structure is slightly more complex than it may seem to. When the ASN.1 structure is freed, all the members of the structure and their submembers etc etc are recursively freed too. But it might not be feasible to free the structure itself. Consider the following case:

```
struct my_figure {       /* The custom structure */
    int flags;           /* <some custom member> */
    /* The type is generated by the ASN.1 compiler */
    Rectangle_t rect;
    /* other members of the structure */
};
```

In this example, the application programmer defined a custom structure with one ASN.1-derived member (rect). This member is not a reference to the Rectangle_t, but an in-place inclusion of the Rectangle_t structure. If the freeing is necessary, the usual procedure of freeing everything must not be applied to the &rect pointer itself, because it does not point to the memory block directly allocated by the memory allocation routine, but instead lies within a block allocated for the my_figure structure.

To solve this problem, the free_struct routine has the additional argument (besides the obvious type descriptor and target structure pointers), which is the flag specifying whether the outer pointer itself must be freed (0, default) or it should be left intact (non-zero value).

```
/* 1. Rectangle_t is defined within my_figure */
struct my_figure {
    Rectangle_t rect;
} *mf = ...;
/*
 * Freeing the Rectangle_t
 * without freeing the mf->rect area
 */
asn_DEF_Rectangle.free_struct(
    &asn_DEF_Rectangle, &mf->rect, 1 /* !free */);


/* 2. Rectangle_t is a stand-alone pointer */
Rectangle_t *rect = ...;
/*
 * Freeing the Rectangle_t
 * and freeing the rect pointer
 */
asn_DEF_Rectangle.free_struct(
    &asn_DEF_Rectangle, rect, 0 /* free the pointer too */);
```

It is safe to invoke the *free_struct* function with the target structure pointer set to 0 (NULL), the function will do nothing.

Chapter 3

Step by step examples

3.1 A ”Rectangle” Encoder

This example will help you to create a simple BER and XER encoder of a ”Rectangle” type used throughout this document.

1. Create a file named **rectangle.asn1** with the following contents:

   ```
   RectangleModule1 DEFINITIONS ::=
   BEGIN

   Rectangle ::= SEQUENCE {
       height  INTEGER,
       width   INTEGER
   }

   END
   ```

2. Compile it into the set of .c and .h files using asn1c compiler [ASN1C]:

   ```
   asn1c -fnative-types rectangle.asn1
   ```

3. Alternatively, use the Online ASN.1 compiler [AONL] by uploading the **rectangle.asn1** file into the Web form and unpacking the produced archive on your computer.

4. By this time, you should have gotten multiple files in the current directory, including the **Rectangle.c** and **Rectangle.h**.

5. Create a main() routine which creates the Rectangle_t structure in memory and encodes it using BER and XER encoding rules. Let’s name the file **main.c**:

```
#include <stdio.h>
#include <sys/types.h>
#include <Rectangle.h>   /* Rectangle ASN.1 type  */

/*
 * This is a custom function which writes the
 * encoded output into some FILE stream.
 */
static int
write_out(const void *buffer, size_t size, void *app_key) {
    FILE *out_fp = app_key;
    size_t wrote;

    wrote = fwrite(buffer, 1, size, out_fp);

    return (wrote == size) ? 0 : -1;
}

int main(int ac, char **av) {
    Rectangle_t *rectangle; /* Type to encode        */
    asn_enc_rval_t ec;      /* Encoder return value  */

    /* Allocate the Rectangle_t */
    rectangle = calloc(1, sizeof(Rectangle_t)); /* not malloc! */
    if(!rectangle) {
      perror(”calloc() failed”);
      exit(71); /* better, EX_OSERR */
    }

    /* Initialize the Rectangle members */
    rectangle->height = 42;  /* any random value */
    rectangle->width  = 23;  /* any random value */

    /* BER encode the data if filename is given */
    if(ac < 2) {
      fprintf(stderr, ”Specify filename for BER output\n”);
    } else {
      const char *filename = av[1];
      FILE *fp = fopen(filename, ”wb”);   /* for BER output */

      if(!fp) {
        perror(filename);
        exit(71); /* better, EX_OSERR */
      }

      /* Encode the Rectangle type as BER (DER) */
      ec = der_encode(&asn_DEF_Rectangle,
            rectangle, write_out, fp);
      fclose(fp);
```

```
            if(ec.encoded == -1) {
              fprintf(stderr,
                ”Could not encode Rectangle (at %s)\n”,
                ec.failed_type ? ec.failed_type->name : ”unknown”);
              exit(65); /* better, EX_DATAERR */
            } else {
              fprintf(stderr, ”Created %s with BER encoded Rectangle\n”,
                filename);
            }
          }

          /* Also print the constructed Rectangle XER encoded (XML) */
          xer_fprint(stdout, &asn_DEF_Rectangle, rectangle);

          return 0; /* Encoding finished successfully */
      }
```

6. Compile all files together using C compiler (varies by platform):

   ```
   cc -I. -o rencode *.c
   ```

7. Voila! You have just created the BER and XER encoder of a Rectangle type, named **rencode**!

3.2 A ”Rectangle” Decoder

This example will help you to create a simple BER decoder of a simple ”Rectangle” type used throughout this document.

1. Create a file named **rectangle.asn1** with the following contents:

```
RectangleModule1 DEFINITIONS ::=
BEGIN

Rectangle ::= SEQUENCE {
    height  INTEGER,
    width   INTEGER
}

END
```

2. Compile it into the set of .c and .h files using asn1c compiler [ASN1C]:

```
asn1c -fnative-types rectangle.asn1
```

3. Alternatively, use the Online ASN.1 compiler [AONL] by uploading the **rectangle.asn1** file into the Web form and unpacking the produced archive on your computer.

4. By this time, you should have gotten multiple files in the current directory, including the **Rectangle.c** and **Rectangle.h**.

5. Create a main() routine which takes the binary input file, decodes it as it were a BER-encoded Rectangle type, and prints out the text (XML) representation of the Rectangle type. Let’s name the file **main.c**:

```
#include <stdio.h>
#include <sys/types.h>
#include <Rectangle.h>   /* Rectangle ASN.1 type  */

int main(int ac, char **av) {
    char buf[1024];      /* Temporary buffer      */
    Rectangle_t *rectangle = 0; /* Type to decode */
    asn_dec_rval_t rval; /* Decoder return value  */
    FILE *fp;            /* Input file handler    */
    size_t size;         /* Number of bytes read  */
    char *filename;      /* Input file name */

    /* Require a single filename argument */
    if(ac != 2) {
      fprintf(stderr, ”Usage: %s <file.ber>\n”, av[0]);
      exit(64); /* better, EX_USAGE */
    } else {
      filename = av[1];
    }

    /* Open input file as read-only binary */
    fp = fopen(filename, ”rb”);
    if(!fp) {
      perror(filename);
      exit(66); /* better, EX_NOINPUT */
    }

    /* Read up to the buffer size */
    size = fread(buf, 1, sizeof(buf), fp);
    fclose(fp);
    if(!size) {
      fprintf(stderr, ”%s: Empty or broken\n”, filename);
      exit(65); /* better, EX_DATAERR */
    }

    /* Decode the input buffer as Rectangle type */
    rval = ber_decode(0, &asn_DEF_Rectangle,
      (void **)&rectangle, buf, size);
    if(rval.code != RC_OK) {
      fprintf(stderr,
        ”%s: Broken Rectangle encoding at byte %ld\n”,
        filename, (long)rval.consumed);
      exit(65); /* better, EX_DATAERR */
    }

    /* Print the decoded Rectangle type as XML */
    xer_fprint(stdout, &asn_DEF_Rectangle, rectangle);

    return 0; /* Decoding finished successfully */
}
```

6. Compile all files together using C compiler (varies by platform):

   ```
   cc -I. -o rdecode *.c
   ```

7. Voila! You have just created the BER decoder of a Rectangle type, named **rdecode**!

Chapter 4

Constraint validation examples

This chapter shows how to define ASN.1 constraints and use the generated validation code.

4.1 Adding constraints into ”Rectangle” type

This example shows how to add basic constraints to the ASN.1 specification and how to invoke the constraints validation code in your application.

1. Create a file named **rectangle.asn1** with the following contents:

```
RectangleModuleWithConstraints DEFINITIONS ::=
BEGIN

Rectangle ::= SEQUENCE {
    height  INTEGER (0..100), -- Value range constraint
    width   INTEGER (0..MAX)  -- Makes width non-negative
}

END
```

2. Compile the file according to procedures shown in the previous chapter.

3. Modify the Rectangle type processing routine (you can start with the main() routine shown in the Section 3.2 on page 22) by placing the following snippet of code *before* encoding and/or *after* decoding the Rectangle type[1]:

[1] Placing the constraint checking code *before* encoding helps to make sure you know the data is correct and within constraints before sharing the data with anyone else.

Placing the constraint checking code *after* decoding, but before any further action depending on the decoded data, helps to make sure the application got the valid contents before making use of it.

```
int ret;           /* Return value */
char errbuf[128];  /* Buffer for error message */
size_t errlen = sizeof(errbuf);  /* Size of the buffer */

/* ... here may go Rectangle decoding code ... */

ret = asn_check_constraints(&asn_DEF_Rectangle,
        rectangle, errbuf, &errlen);
/* assert(errlen < sizeof(errbuf)); // you may rely on that */
if(ret) {
        fprintf(stderr, ”Constraint validation failed: %s\n”,
          errbuf   /* errbuf is properly nul-terminated */
        );
        /* exit(...); // Replace with appropriate action */
}

/* ... here may go Rectangle encoding code ... */
```

4. Compile the resulting C code as shown in the previous chapters.

5. Try to test the constraints checking code by assigning integer value 101 to the **.height** member of the Rectangle structure, or a negative value to the **.width** member. In either case, the program should print ”Constraint validation failed” message, followed by the short explanation why validation did not succeed.

6. Done.

Part II

ASN.1 Basics

Chapter 5

Abstract Syntax Notation: ASN.1

This chapter defines some basic ASN.1 concepts and describes several most widely used types. It is by no means an authoritative or complete reference. For more complete ASN.1 description, please refer to Olivier Dubuisson's book [Dub00] or the ASN.1 body of standards itself [ITU-T/ASN.1].

The Abstract Syntax Notation One is used to formally describe the semantics of data transmitted across the network. Two communicating parties may have different formats of their native data types (i.e. number of bits in the integer type), thus it is important to have a way to describe the data in a manner which is independent from the particular machine's representation. The ASN.1 specifications are used to achieve the following:

- The specification expressed in the ASN.1 notation is a formal and precise way to communicate the data semantics to human readers;
- The ASN.1 specifications may be used as input for automatic compilers which produce the code for some target language (C, C++, Java, etc) to encode and decode the data according to some encoding rules (which are also defined by the ASN.1 standard).

Consider the following example:

```
Rectangle ::= SEQUENCE {
    height  INTEGER,
    width   INTEGER
}
```

This ASN.1 specification describes a constructed type, *Rectangle*, containing two integer fields. This specification may tell the reader that there exists this kind of data structure and that some entity may be prepared to send or receive it. The question on *how* that entity is going to send or receive the *encoded data* is outside the scope of

ASN.1. For example, this data structure may be encoded according to some encoding rules and sent to the destination using the TCP protocol. The ASN.1 specifies several ways of encoding (or ”serializing”, or ”marshaling”) the data: BER, PER, XER and others, including CER and DER derivatives from BER.

The complete specification must be wrapped in a module, which looks like this:

```
RectangleModule1
    { iso org(3) dod(6) internet(1) private(4)
      enterprise(1) spelio(9363) software(1)
      asn1c(5) docs(2) rectangle(1) 1 }
    DEFINITIONS AUTOMATIC TAGS ::=
BEGIN

-- This is a comment which describes nothing.
Rectangle ::= SEQUENCE {
    height  INTEGER,        -- Height of the rectangle
    width   INTEGER         -- Width of the rectangle
}

END
```

The module header consists of module name (RectangleModule1), the module object identifier ({...}), a keyword ”DEFINITIONS”, a set of module flags (AUTOMATIC TAGS) and ”::= BEGIN”. The module ends with an ”END” statement.

5.1 Some of the ASN.1 Basic Types

5.1.1 The BOOLEAN type

The BOOLEAN type models the simple binary TRUE/FALSE, YES/NO, ON/OFF or a similar kind of two-way choice.

5.1.2 The INTEGER type

The INTEGER type is a signed natural number type without any restrictions on its size. If the automatic checking on INTEGER value bounds are necessary, the subtype constraints must be used.

```
SimpleInteger ::= INTEGER

-- An integer with a very limited range
SmallPositiveInt ::= INTEGER (0..127)

-- Integer, negative
NegativeInt ::= INTEGER (MIN..0)
```

5.1.3 The ENUMERATED type

The ENUMERATED type is semantically equivalent to the INTEGER type with some integer values explicitly named.

```
FruitId ::= ENUMERATED { apple(1), orange(2) }

-- The numbers in braces are optional,
-- the enumeration can be performed
-- automatically by the compiler
ComputerOSType ::= ENUMERATED {
    FreeBSD,          -- acquires value 0
    Windows,          -- acquires value 1
    Solaris(5),       -- remains 5
    Linux,            -- becomes 6
    MacOS             -- becomes 7
}
```

5.1.4 The OCTET STRING type

This type models the sequence of 8-bit bytes. This may be used to transmit some opaque data or data serialized by other types of encoders (i.e. video file, photo picture, etc).

5.1.5 The OBJECT IDENTIFIER type

The OBJECT IDENTIFIER is used to represent the unique identifier of any object, starting from the very root of the registration tree. If your organization needs to uniquely identify something (a router, a room, a person, a standard, or whatever), you are encouraged to get your own identification subtree at http://www.iana.org/protocols/forms.htm.

For example, the very first ASN.1 module in this Chapter (RectangleModule1) has the following OBJECT IDENTIFIER: 1 3 6 1 4 1 9363 1 5 2 1 1.

```
ExampleOID ::= OBJECT IDENTIFIER

rectangleModule1-oid ExampleOID
  ::= { 1 3 6 1 4 1 9363 1 5 2 1 1 }

-- An identifier of the Internet.
internet-id OBJECT IDENTIFIER
  ::= { iso(1) identified-organization(3)
        dod(6) internet(1) }
```

As you see, names are optional.

5.1.6 The RELATIVE-OID type

The RELATIVE-OID type has the semantics of a subtree of an OBJECT IDENTIFIER. There may be no need to repeat the whole sequence of numbers from the root of the registration tree where the only thing of interest is some of the tree’s subsequence.

```
this-document RELATIVE-OID ::= { docs(2) usage(1) }

this-example RELATIVE-OID ::= {
    this-document assorted-examples(0) this-example(1) }
```

5.2 Some of the ASN.1 String Types

5.2.1 The IA5String type

This is essentially the ASCII, with 128 character codes available (7 lower bits of an 8-bit byte).

5.2.2 The UTF8String type

This is the character string which encodes the full Unicode range (4 bytes) using multi-byte character sequences.

5.2.3 The NumericString type

This type represents the character string with the alphabet consisting of numbers (”0” to ”9”) and a space.

5.2.4 The PrintableString type

The character string with the following alphabet: space, ”’” (single quote), ”(”, ”)”, ”+”, ”,” (comma), ”-”, ”.”, ”/”, digits (”0” to ”9”), ”:”, ”=”, ”?”, upper-case and lower-case letters (”A” to ”Z” and ”a” to ”z”).

5.2.5 The VisibleString type

The character string with the alphabet which is more or less a subset of ASCII between the space and the ”~” symbol (tilde).

Alternatively, the alphabet may be described as the PrintableString alphabet presented earlier, plus the following characters: ”!”, ”””, ”#”, ”$”, ”%”, ”&”, ”*”, ”;”, ”<”, ”>”, ”[”, ”\”, ”]”, ”^”, ”_”, ”‘” (single left quote), ”{”, ”|”, ”}”, ”~”.

5.3 ASN.1 Constructed Types

5.3.1 The SEQUENCE type

This is an ordered collection of other simple or constructed types. The SEQUENCE constructed type resembles the C ”struct” statement.

```
Address ::= SEQUENCE {
    -- The apartment number may be omitted
    apartmentNumber      NumericString OPTIONAL,
    streetName           PrintableString,
    cityName             PrintableString,
    stateName            PrintableString,
    -- This one may be omitted too
    zipNo                NumericString OPTIONAL
}
```

5.3.2 The SET type

This is a collection of other simple or constructed types. Ordering is not important. The data may arrive in the order which is different from the order of specification. Data is encoded in the order not necessarily corresponding to the order of specification.

5.3.3 The CHOICE type

This type is just a choice between the subtypes specified in it. The CHOICE type contains at most one of the subtypes specified, and it is always implicitly known which choice is being decoded or encoded. This one resembles the C ”union” statement.

The following type defines a response code, which may be either an integer code or a boolean ”true”/”false” code.

```
ResponseCode ::= CHOICE {
    intCode    INTEGER,
    boolCode   BOOLEAN
}
```

5.3.4 The SEQUENCE OF type

This one is the list (array) of simple or constructed types:

```
-- Example 1
ManyIntegers ::= SEQUENCE OF INTEGER

-- Example 2
ManyRectangles ::= SEQUENCE OF Rectangle

-- More complex example:
```

```
-- an array of structures defined in place.
ManyCircles ::= SEQUENCE OF SEQUENCE {
                            radius INTEGER
                            }
```

5.3.5 The SET OF type

The SET OF type models the bag of structures. It resembles the SEQUENCE OF type, but the order is not important: i.e. the elements may arrive in the order which is not necessarily the same as the in-memory order on the remote machines.

```
-- A set of structures defined elsewhere
SetOfApples :: SET OF Apple

-- Set of integers encoding the kind of a fruit
FruitBag ::= SET OF ENUMERATED { apple, orange }
```

Bibliography

[ASN1C] The Open Source ASN.1 Compiler. http://lionet.info/asn1c

[AONL] Online ASN.1 Compiler. http://lionet.info/asn1c/asn1c.cgi

[Dub00] Olivier Dubuisson — *ASN.1 Communication between heterogeneous systems* — Morgan Kaufmann Publishers, 2000. http://asn1.elibel.tm.fr/en/book/. ISBN:0-12-6333361-0.

[ITU-T/ASN.1] ITU-T Study Group 17 – Languages for Telecommunication Systems http://www.itu.int/ITU-T/studygroups/com17/languages/