Snacc 1.0: A High Performance ASN.1 to C/C++ Compiler

Michael Sample and Gerald Neufeld
msample@cs.ubc.ca neufeld@cs.ubc.ca

Department of Computer Science
University of British Columbia
6356 Agricultural Rd.
Vancouver, British Columbia
Canada, V6T 1Z2

February 1993

This work was made possible by grants from the Canadian Institute for Telecommunications Research (CITR) and Natural Sciences and Engineering Research Council of Canada (NSERC).

Contents

1 Introduction **5**
1.1 Installing snacc 6
1.2 Running snacc 7
1.3 Reporting Bugs 10

2 Compiler Design **11**
2.1 Overview 11
2.2 Pass 1: Parsing the Useful Types Module 12
2.3 Pass 2: Parsing ASN.1 Modules 13
2.4 Pass 3: Linking Types 14
2.5 Pass 4: Parsing Values 15
2.6 Pass 5: Linking Values 16
2.7 Pass 6: Processing Macros 17
2.8 Pass 7: Normalizing Types 18
2.9 Pass 8: Marking Recursive Types 19
2.10 Pass 9: Semantic Error Checking 19
2.11 Pass 10: Generating C/C++ Type Information 20
2.12 Pass 11: Sorting Types 22
2.13 Pass 12: Generating Code 24

3 C Code Generation **25**
3.1 Introduction 25
3.2 ASN.1 to C Naming Conventions 26
3.3 ASN.1 to C Data Structure Translation 27
3.4 Encode Routines 28
3.5 Decode Routines 29
3.6 Print Routines 31
3.7 Free Routines 32
3.8 ASN.1 to C Value Translation 32
3.9 Compiler Directives 33
3.10 Compiling the Generated C Code 38

4 C ASN.1 Library **39**
4.1 Overview 39
4.2 Tags 39

4.3 Lengths 40
4.4 BOOLEAN 41
4.5 INTEGER 42
4.6 NULL 43
4.7 REAL 44
4.8 BIT STRING 44
4.9 OCTET STRING 46
4.10 OBJECT IDENTIFIER 47
4.11 SET OF and SEQUENCE OF 48
4.12 ANY and ANY DEFINED BY 49
4.13 Buffer Management 53
4.13.1 Buffer Reading Routine Semantics 54
4.13.2 Buffer Writing Routine Semantics 55
4.13.3 Buffer Configuration 56
4.13.4 ExpBuf Buffers 57
4.13.5 SBuf Buffers 59
4.13.6 MinBuf Buffers 60
4.13.7 Hybrid Buffer Solutions 61
4.14 Dynamic Memory Management 61
4.15 Error Management 63

5 C++ Code Generation **65**
5.1 Introduction 65
5.2 ASN.1 to C++ Naming Conventions 66
5.3 ASN.1 to C++ Class Translation 66
5.3.1 SET and SEQUENCE 68
5.3.2 CHOICE 69
5.3.3 SET OF and SEQUENCE OF 70
5.3.4 ENUMERATED, Named Numbers and Named Bits 74
5.4 ASN.1 to C++ Value Translation 75
5.5 Compiler Directives 75
5.6 Compiling the Generated C++ Code 76

6 C++ ASN.1 Library **77**
6.1 Overview 77
6.2 Tags 78
6.3 Lengths 78
6.4 The AsnType Base Class 78
6.5 BOOLEAN 79
6.6 INTEGER 81
6.7 NULL 81
6.8 REAL 82
6.9 BIT STRING 83
6.10 OCTET STRING 85
6.11 OBJECT IDENTIFIER 87

6.12 SET OF and SEQUENCE OF 88
6.13 ANY and ANY DEFINED BY 89
6.14 Buffer Management 91
6.15 Dynamic Memory Management 92
6.16 Error Management 92

7 Modifying the Compiler 93

8 Future Work 94

A The Module Data Structure ASN.1 Definition 96

Chapter 1

Introduction

Snacc compiles ASN.1 [1] (Abstract Syntax Notation One) modules into C or C++. The generated C or C++ code contains equivalent data structures and routines to convert values between the internal (C or C++) representation and the corresponding BER [2] (Basic Encoding Rules) format. The name "snacc" is an acronym for "Sample Neufeld ASN.1 to C/C++ Compiler".

The ASN.1 data structure language can specify complex types such as lists and recursively defined types. BER data values are defined independently of any computer architecture, providing a universal data value representation that is useful for sharing data in heterogeneous networks.

The process of converting an ASN.1 value from its C or C++ representation into an equivalent BER data value is called encoding and the reverse process is called decoding. This document was written assuming that the reader is familiar ASN.1 and BER. Further information on ASN.1 and BER can be found in [7], [4], [1] and [2].

Compiling ASN.1 into C is not a new idea but many other tools such as UBC's CASN1 [5], ISODE's PEPY/POSY [6], and commercial tools either do not parse ASN.1 '90, produce slow encoders and decoders or are outrageously expensive. The aim of this tool is to provide an ASN.1 compiler that parses ASN.1 '90, produces efficient encoding and decoding routines and is freely available. Effort has been made to make the generated encoders and decoders relatively easy to fit into different software environments.

Some of snacc's features include:

- parses CCITT ASN.1 '90 including subtype notation
- can compile and link inter-dependent ASN.1 modules (IMPORTS/EXPORTS)
- some X.400 and SNMP macros are parsed
- macro *definitions* do not generate syntax errors but are are not processed. The macro definitions are retained as a string internally (if you want to modify the compiler to

process them).

- value notation is parsed. OBJECT IDENTIFIERs, INTEGERs and BOOLEANs are translated to C/C++ values. Any other value in {}'s is kept as a string internally (if you want to modify the compiler to process them).
- optionally supports ";" separated type or value definitions in the ASN.1 source. This is useful for dealing with some macros and other language ambiguities that introduce parsing problems.
- ANY DEFINED BY types are supported via the SNMP OBJECT-TYPE macro

1.1 Installing snacc

First of all, if you haven't already done so, un-archive snacc to produce the directory *snacc* and its contents. The following tools are required to compile snacc:

- `lex` or GNU's `flex` (`flex` is recommended)
- `yacc` or GNU's `bison` (`bison` is recommended)
- a C compiler and `make`

Some versions of `yacc` will choke due to the large size of the `asn1.yacc` file, however, I have had no problems with `bison`. Our `yacc` grammar for ASN.1 has 61 shift/reduce errors and 2 reduce/reduce errors. Most of these errors were introduced when certain macros were added to the compiler. Some of the shift/reduce errors will require you to follow the offending macro in the ASN.1 module with a semi-colon. The reduce/reduce errors were introduced by macros that have "Type or Value Lists" because the NULL Type and NULL values use the same symbol, "NULL". This is not a problem since no real processing is done with the macros in question at the present.

`Lex` will work for the `asn1.lex` file but `flex` will typically produce a smaller executable. Most versions of `lex` have a small maximum token size that will cause problems for long tokens in the ASN.1 source files, such as quoted strings. To avoid this problem, increase the `YYLMAX` value in the generated *lex.yy.c* file to at least 2048. `Flex` does not seem to have this problem.

The compiler and library C code has been written to support ANSI or non-ANSI C. ANSI C is used by default; this can be configured in `snacc/c_include/asn_config.h`.

By default, the compiler's makefiles use `flex`, `bison` and `gcc`. If you wish to change these, edit the following files:

```
snacc/src/makefile
snacc/src/back_ends/c_gen/makefile
snacc/src/back_ends/c++_gen/makefile
```

The C runtime library uses `gcc`, and its makefile is `snacc/c_lib/makefile`. The C++ runtime library uses `g++` (`gcc-2.2.3`) and its makefile is `snacc/c++_lib/makefile`.

Finally, to compile `snacc` and the C and C++ runtime libraries, type the following at the shell prompt:

```
%1 cd snacc
%2 make both
```

If you wish to install only the C or only the C++ versions of the library, type `make c` or `make c++`, respectively, instead of `make both`. If the make succeeds, the snacc binary, *snacc*, should be in the `snacc/bin/` directory, the C runtime libraries, *libasn1csbuf.a*, *libasn1cebuf.a*, and *libasn1cmbuf.a*, should be in the `snacc/c_lib` and the C++ runtime library, *libasn1c++.a* should be in the `snacc/c++_lib`. The `.o` and other junk files will have been removed.

After compiling the libraries, you can test the library routines with `snacc/c_examples/test_lib/test_lib` or `snacc/c++_examples/test_lib/test_lib`. These programs run simple encoding and decoding tests on all of the library types. You can test the snacc compiler with the other examples.

A manual page that contains information on running snacc can be found in `snacc/doc/snacc.1`. This should be installed in section 1 of the manual. You can use `nroff -man snacc.1` to view it if you don't want to install it.

1.2 Running snacc

Snacc is typically invoked from the shell command line an has the synopsis:

```
snacc [-h] [-c] [-C] [-P] [-t] [-e] [-d] [-p] [-f]
      [-u <useful types ASN.1 file>] [-mf <max file name len>]
      [-l <neg number>] <ASN.1 file list>
```

Snacc generates C or C++ source code for BER encode and decode routines as well as print and free routines for each type in the given ASN.1 modules. Most of the 1990 ASN.1 features are parsed although some do not affect the generated code. Fairly rigourous error checking is performed on the ASN.1 source; any errors detected will be reported.

Each file in the ASN.1 file list should contain a complete ASN.1 module. ASN.1 modules that use the IMPORTS feature must be compiled together (specify all necessary modules in the ASN.1 file list). The generated source files will include each module's header file in the command line order. This makes it important to order the modules from least dependent to most dependent on the command line to avoid type ordering problems. Currently, snacc assumes that each ASN.1 file given on the command line depends on all of the others on the command line. No attempt is made to only include the header files from modules referenced in the import list for that module.

If the target language is C, snacc will generate a `.h` and `.c` file for each specified ASN.1 module. If the target language is C++, snacc will generate a `.h` and `.C` file for each module. The generated file names will be derived from the module names.

The command line options are:

-h short for "help", prints a synopsis of snacc and exits.

-c causes snacc to generate C source code. This is the default behaviour of snacc if neither of the `-c` or `-C` options are given. Only one of `-c` or `-C` should be specified.

-C causes snacc to generate C++ source code.

-P causes snacc to print the parsed ASN.1 modules to `stdout` after the types have been linked, sorted, and processed. This option is useful for debugging snacc and observing the modifications snacc performs on the types to make code generation simpler.

The options, `-t, -v, -e, -d, -p,` and `-f` affect what types and routines go into the generated source code. If none of them are given on the command line, snacc assumes that all of them are in effect. For example, if you do not need the Free or Print routines, you should give the `-t -v -e -d` options to snacc. This lets you trim the size of the generated code by removing unnecessary routines; the code generated from large ASN.1 specifications can produce very large binaries.

-t causes snacc to generate type definitions in the target language for each ASN.1 type.

-v causes snacc to generate value definitions in the target language for each ASN.1 value. Currently value definitions are limited to INTEGERs, BOOLEANs and OBJECT IDENTIFIERs.

-e causes snacc to generate encode routines in the target language for each ASN.1 type.

-d causes snacc to generate decode routines in the target language for each ASN.1 type.

-p causes snacc to generate print routines in the target language for each ASN.1 type.

-f causes snacc to generate free routines in the target language for each ASN.1 type. This option only works when the target language is C. The free routines hierarchically free C values. A more efficient approach is to use the provided nibble-memory system. The nibble memory permits freeing an entire decoded value without traversing the decoded value. This is the default memory allocator used by snacc generated decoders. See file `snacc/c_include/asn_config.h` to change the default memory system. For more information on the memory management see Section 4.14.

-u *file* causes snacc to read the useful types definitions from the ASN.1 module in file *file* for linking purposes. For some ASN.1 specifications, such as SNMP, the useful types are not needed. The types in the given useful types file are globally available to all modules; a useful type definition is overridden by a local or explicitly imported type with the same name. The useful type module can be found in `snacc/asn1specs/useful.asn1` and contains:

- ObjectDesccriptor
- NumericString
- PrintableString
- TeletexString
- T61String
- VideoTexString
- IA5String
- GraphicString
- ISO646String
- GeneralString
- UTCTime
- GeneralizedTime
- EXTERNAL

-mf *number* causes the names of the generated source files to have a maximum length of *number* characters, including their suffix. The *number* argument must be at least 3. This option is useful for supporting operating systems that only support short file names. A better solution is to shorten the module name of each ASN.1 module.

-l *number* this is fairly obscure but may be useful. Each error that the decoders can report is given an id number. The number *number* is where the error ids start decreasing from as they are assigned to errors. The default is -100 if this option is not given. Avoid using a number in the range -100 to 0 since they may conflict with the library routines' error ids. If you are re-compiling the useful types for the library use -50. Another use of this option is to integrate newly generated code with older code; if done correctly, the error ids will not conflict.

Since ASN.1 has different scoping rules than C and C++, some name munging is done for types, named-numbers etc. to eliminate conflicts. Some capitalization schemes where chosen to fit common C programming style. For all names, dashes in the ASN.1 source are converted to underscores. See Sections 3.2 and 5.2 for more naming information.

The module name is used as a base name for the generated source file names. It will be put into lowercase and dashes will be replaced with underscores. Module names that result in file names longer than specified with the `-mf` option will be truncated. If the `-mf` option was not given, file names will be truncated if they are too long for the target file system. You may want to shorten long module names to meaningful abbreviations. This will avoid file name conflicts for module names that are truncated to the same substring. Any module name and file name conflicts will be reported.

If your ASN.1 modules have syntactic or semantic errors, each error will be printed to `stderr` along with the file name and line number of where it occurred. These errors are usable by GNU emacs compiling tools. See the next chapter for more information on the types of errors snacc can detect.

More errors can be detected and reported in a single compile if type and value definitions are separated by semi-colons. Separating type and value definitions with semi-colons is not required, and if used, need not be used to separate all type and value definitions. Semi-colons are necessary after some macros that introduce ambiguity. In general, if you get a parse error you can't figure out, try separating the surrounding type/value definitions with semi-colons.

1.3 Reporting Bugs

This is the first release of snacc so bugs are surely present. If you find some bugs or have other comments, please send email to snacc-bugs@cs.ubc.ca.

Please include the offending ASN.1 source, the command line options you were using and the hardware and operating system configuration.

We will respond to bug report as time permits.

Chapter 2

Compiler Design

2.1 Overview

The snacc compiler is implemented with `yacc`, `lex` (actually GNU's equivalents, `bison` and `flex`) and `C`. Despite the shortcomings of `lex` and `yacc`, they provide reasonable performance without too much programming effort. Since `yacc` parsers are extremely difficult to modify during runtime, any macro that you want the compiler to handle must be hand coded into the ASN.1 `yacc` grammar (`snacc/src/asn1.yacc`) followed by recompilation of snacc. Macro definitions do not need special consideration since they are skipped by the compiler. Macro definitions and complex value notation are kept as text in the data structure resulting from a parse if you want to try to parse and process them.

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

1 parse useful types ASN.1 module

2 parse all user specified ASN.1 modules

3 link local and imported type references in all modules

4 parse values in all modules

5 link local and imported value references in all modules

6 process any macro types

7 normalize types

8 mark recursive types and signal any recursion related errors

9 check for semantic errors in all modules

10 generate C/C++ type information for each ASN.1 type

11 Sort the types from least dependent to most dependent

12 generate the C/C++

The source code for the compiler resides in `snacc/src/` and the back ends are in `snacc/src/back_ends/c_gen/` and `snacc/src/back_ends/c++_gen/`.

2.2 Pass 1: Parsing the Useful Types Module

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

The useful types module should be passed to snacc with the `-u` flag in front of it. The `-u` flag tells snacc to treat the module in a special way. Instead of parsing the module and generating code for it, snacc parses the module and makes the types in it accessible to all of the other modules being parsed. Note that the other modules do not need to explicitly import from the useful types module. See Section 2.4 for more information on how useful types affect linking.

The encode, decode, and other routines for the useful types are in the runtime library. Currently, the useful types library routines are the same as the ones the compiler would normally generate given the useful types module. However, since they are in the library, you can modify them to check character sets (string types), or convert local time formats into their BER equivalent (UTCTime, GeneralizedTime).

The following types are in the useful types module:

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

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

If you use the EXTERNAL type, you must provide the mechanism to encode and decode the value in the embedded CHOICE, `encoding`. The type and transfer syntax of the value in an EXTERNAL type is not known when the ASN.1 code is compiled by snacc. Snacc cannot generate encoders and decoders without complete type information and only supports a single set of encoding rules, BER.

2.3 Pass 2: Parsing ASN.1 Modules

During this pass, all of the specified modules are parsed into the *Module* data structure. The ASN.1 source files are not consulted again, after they are parsed. `Yacc` and `lex` are doing the work in this step. (see files `snacc.c`, `asn1.yacc`, `asn1.lex` and `asn1module.h`).

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

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

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

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

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

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

2.4 Pass 3: Linking Types

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

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

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

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

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

Next, each module's import *lists* are resolved by finding the named module and then verifying that the named module contains all of the imported types.

Then for each module, each type reference (except those of the form *modulename.typename*) is assumed to be a local type reference and the linker attempts to find a local type definition of the same name to resolve it with. If a matching local definition is found, the type reference is resolved and the linker continues with the next type reference.

For each type reference of the form *modulename.typename*, the linker looks in the module with name *modulename* for the type *typename*. If the type is found the reference is resolved, otherwise an linking error is reported. Note that this form of type reference provides a special scope that does not conflict with other local or imported types in that module.

For type references that failed to resolve locally and are not of the form *modulename.typename*, the linker looks in the import lists of the current type reference's module for a type to resolve with. If the type is found in the import lists, the reference is resolved.

For the remaining unresolved type references (failed local and legal import resolution and are not of the form *modulename.typename*), the linker looks in the useful types module, if one was specified with the `-u` option. If the type is found in the useful types module then the reference is resolved, otherwise a linking error is reported.

Note that when a useful types module is specified, it is globally available to all modules, but it has the lowest linking priority. That is, if a type reference can be resolved legally without the useful types module, it will be.

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

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

2.5 Pass 4: Parsing Values

The fourth pass attempts to parse any value is enclosed in {}'s in the given modules. INTEGERS, REALs and BOOLEANS that are not enclosed in braces are parsed in the first pass.

The value parser is implemented without `yacc` and `lex` and uses each value's type information to help parse the value. Values within {}'s hidden within types such as default values and parts of subtypes are not parsed. Since subtypes and default values do not affect the generated code, upgrading the value parser in this respect is not very useful.

The only type of value in {}'s that is parsed is the OBJECT IDENTIFIER. All of the OBJECT IDENTIFIER value forms are supported but snacc loosens the restrictions on using arc names defined in the OBJECT IDENTIFIER tree.

ASN.1 allows OBJECT IDENTIFIER values to reference special built-in arc names from the OBJECT IDENTIFIER tree defined in in Annexes B, C and D of X.208. For example the first arc in an OBJECT IDENTIFIER value can be either `ccitt` `iso` or `joint-iso-ccitt`. The acceptable arc names are context dependent; for example the second arc can be one of `standard`, `registration-authority`, `member-body` or `identified-organization` only if the first arc was `iso` or 1.

Snacc uses a simplified algorithm to handle references to the arc names defined in OBJECT IDENTIFIER tree. Any arc value that is represented by a single identifier is checked to see if it is one of the arc names defined in the OBJECT IDENTIFIER tree; context is ignored. If the identifier matches one of the arc names then its value is set accordingly. The lack of context sensitivity in snacc's algorithm may cause the arc name to link with an arc name from the OBJECT IDENTIFIER tree when a local or imported INTEGER was desired.

The following is the list special arc names that snacc understands and their values (see `snacc/src/oid.c`):

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

2.6 Pass 5: Linking Values

The fifth pass links value references. The value linker looks for value references to resolve in value definitions and type definitions, including default values and subtyping information. The value linking algorithm is virtually identical to the type linking pass (see Section 2.4).

Note that snacc allows INTEGER values to be defined in two ways, one by the standard `valuename Type ::= Value` style and the other by a certain form of OBJECT IDENTIFIER arc values. For example, parsing the following OBJECT IDENTIFIER value in pass four will define the INTEGER values `org` as 3 and `dod` as 6 in addition to the `internet` OBJECT IDENTIFIER value, `{1 3 6 1}`.

```
internet OBJECT IDENTIFIER ::= { iso org(3) dod(6) 1 }
```

Here is an example that shows illustrates the OBJECT IDENTIFIER parsing and linking described above. The following values:

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

are equivalent to this:

```
foo OBJECT IDENTIFIER ::= { 2 2 88 28 }
bar OBJECT IDENTIFIER ::= { 2 2 88 28  1 }
```

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

2.7 Pass 6: Processing Macros

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

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

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

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

The following macros are parsed:

- OPERATION (ROS)
- ERROR (ROS)
- BIND (ROS)
- UNBIND (ROS)
- APPLICATION-SERVICE-ELEMENT (ROS)
- APPLICATION-CONTEXT
- EXTENSION (MTSAS)
- EXTENSIONS (MTSAS)
- EXTENSION-ATTRIBUTE (MTSAS)
- TOKEN (MTSAS)
- TOKEN-DATA (MTSAS)
- SECURITY-CATEGORY (MTSAS)

- OBJECT (X.407)
- PORT (X.407)
- REFINE (X.407)
- ABSTRACT-BIND (X.407)
- ABSTRACT-UNBIND (X.407)
- ABSTRACT-OPERATION (X.407)
- ABSTRACT-ERROR (X.407)
- ALGORITHM (X.509)
- ENCRYPTED (X.509)
- PROTECTED (X.509)
- SIGNATURE (X.509)
- SIGNED (X.509)
- OBJECT-TYPE (SNMP)

2.8 Pass 7: Normalizing Types

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

1 COMPONENTS OF types are replaced with the contents of the SET or SEQUENCE components that they reference.

2 SELECTION types are replaced with the type they reference.

3 SEQUENCE, SET, CHOICE, SET OF and SEQUENCE OF *definitions* embedded in other types are made into separate type definitions.

4 for modules in which "IMPLICIT TAGS" is specified, tagged type references such as `[APPLICATION 2] Foo` are marked IMPLICIT if the referenced type (`FOO` in this case) is not an untagged CHOICE or untagged ANY type.

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

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

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

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

2.9 Pass 8: Marking Recursive Types

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

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

```
A ::= B
B ::= C
C ::= A
```

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

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

2.10 Pass 9: Semantic Error Checking

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

- elements of CHOICE and SET types must have distinct tags.
- CHOICE, ANY, and ANY DEFINED BY types cannot be implicitly tagged.
- type and value names within the same scope must be unique.
- field names in a SET, SEQUENCE or CHOICE must be distinct. If a CHOICE is a member of a SET, SEQUENCE or CHOICE and has no field name, then the embedded

CHOICE's field names must be distinct from its parents to avoid ambiguity in value notation.

- an APPLICATION tag code can only be used once per module.
- each value in a named bit list (BIT STRINGs) or named number list (INTEGERs and ENUMERATED) must be unique within its list.
- each identifier in a named bit list or named number list must be unique within its list.
- the tags on a series of one or more consecutive OPTIONAL or DEFAULT SEQUENCE elements and the following element must be distinct.
- gives a warning if an ANY DEFINED BY type appears in a SEQUENCE before its identifier or in a SET. These would allow encodings where the ANY DEFINED BY value was prior to its identifier in the encoded value; ANY DEFINED BY values are difficult to decode without knowing their identifier.

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

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

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

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

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

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

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

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

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

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

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

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

printRoutineName name of this type definition's printing routine.

encodeRoutineName name of this type definition's encoding routine.

decodeRoutineName name of this type definition's decoding routine.

freeRoutineName name of this type definition's freeing routine.

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

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

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

isPtrForTypeDef same as C isPtrForTypeDef.

isPtrForOpt same as C isPtrForOpt.

isPtrInChoice same as C isPtrInChoice

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

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

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

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

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

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

typeName type name of the referenced type.

isPtr whether this reference is by pointer.

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

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

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

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

2.12 Pass 11: Sorting Types

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

Some compilers such as CASN1 [5] require the user to order the types within the ASN.1 modules. This can be tedious and since snacc may generate new type definitions from nested aggregate type definitions in the normalization pass, the user does not have complete control over the order of every type definition. (The user could use the `-P` option to get the normalized ASN.1 and then order it but that is painful as well.)

Snacc attempts to sort the types from least dependent to most dependent using the following convoluted algorithm:

First, separate the type definitions within a module into the groups:

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

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

3 type definitions that reference locally defined types.

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

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

Now we describe how the 3rd group of type definitions is sorted.

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

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

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

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

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

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

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

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

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

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

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

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

translates to the following C data types:

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

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

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

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

2.13 Pass 12: Generating Code

This pass creates and fills the source files with C or C++ code. The purpose of the normalization, sorting and error detection passes is to simplify this pass.

The normalization pass simplified the ASN.1 types in various ways to make C/C++ type and code generation simpler.

The type sorting pass hopefully eliminates type dependency problems in the generated code. The C/C++ type generator simply proceeds through the ordered type list writing the C/C++ type definitions to a header file.

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

The next chapters describe the code that is generated by snacc and the libraries the generated code uses.

Chapter 3

C Code Generation

3.1 Introduction

Snacc was designed primarily to provide high-performance encoders and decoders. Key areas to optimize are buffer and memory management. Buffers are used to hold encoded values and the memory management is used when building the internal representation of value when decoding.

C macros are used where possible to eliminate function call overhead for small, commonly used routines. Using macros with constant expressions as parameters allows smarter C compilers to do some of the calculations at compile time. In general, any short-cuts can be taken without sacrificing the robustness of code are used.

The generated code can be quite large; large reductions in of the size of the binaries can be achieved by using the optimizing options of your C compiler.

We will use an example ASN.1 module, EX1, to help explain snacc's code generation. The EX1 module uses some of the common built-in types and contains some simple values. The field names have been left out to show snacc naming conventions. The C generation code is in `snacc/src/back_ends/c_gen` if you want to alter it.

```
EX1 DEFINITIONS ::=
BEGIN

anOidVal      OBJECT IDENTIFIER ::= { joint-iso-ccitt 40 foobar(29) }
theSameOidVal OBJECT IDENTIFIER ::= { 2 40 29 }
anIntVal      INTEGER ::= 1
aBoolVal      BOOLEAN ::= TRUE

T1 ::= SEQUENCE
{
   INTEGER OPTIONAL,
   OCTET STRING OPTIONAL,
```

```
   ENUMERATED { a(0), b(1), c(2) },
   SEQUENCE OF INTEGER,
   SEQUENCE { id OBJECT IDENTIFIER, value OCTET STRING },
   CHOICE { INTEGER, OBJECT IDENTIFIER }
}

END
```

Use the following command to compile the EX1 ASN.1 module.

```
%1 snacc -u snacc/asn1specs/useful.asn1 snacc/asn1specs/ex1.asn1
```

This will produces the files `ex1.h` and `ex1.c`.

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

3.2 ASN.1 to C Naming Conventions

For any given module snacc may produce C type definitions, functions and `#defines`. We assume that all C `typedef`, `struct`, `enum` and `union` tag, `enum` value, variable, `#define` and function names share a single name space.

The C type name for a type is the same as its ASN.1 type name (with any hyphens converted to underscores) unless there is a conflict. Since, unlike ASN.1, the C types for each ASN.1 module share the same name space, snacc makes sure the C typenames are unique amoung all the modules and that they do not conflict with C keywords. The conflicts are resolved by appending digits to the conflicting name. To avoid confusing numbered type names etc., you should edit the ASN.1 source and name them properly.

Named numbers, ENUMERATED values and named bits are put in entirely in upper case to match the common C convention for `#define` and `enum` values.

Empty field names in SETs, SEQUENCEs, and CHOICEs will be filled. The field name is derived from the type name for that field. The library types such as INTEGER etc. have default field names defined by the compiler (see `snacc/src/back_ends/c_gen/rules.c` and `snacc/src/back_ends/cpp_rules.c`). The first letter of the field name is in lower case. Again, empty field names should be fixed properly by adding them to the ASN.1 source.

New type definitions will be generated for SETs, SEQUENCEs, CHOICEs, ENUMERATED, INTEGERs with named numbers and BIT STRING with named bits whose definitions are embedded in other SET, SEQUENCE, SET OF, SEQUENCE OF, or CHOICE definitions. The name of the new type is derived from the name of the type in which it was embedded. Perhaps a better way would use the field name as well, if present.

3.3 ASN.1 to C Data Structure Translation

To handle the different scoping rules between ASN.1 and C, some the names of some ASN.1 data structure elements such as ENUMERATED type symbols may be altered to avoid conflicts. The T1 type in example ASN.1 module EX1 has no field names so snacc will generate them. It is recommended to provide field names in the ASN.1 source instead of relying on compiler generated names. The following is the generated C data structure for the EX1 module from the `ex1.h` file (function prototypes have been removed):

```
typedef enum
{
    A = 0,
    B = 1,
    C = 2
} T1Enum; /* ENUMERATED { A(0), B(1), C(2) }  */

typedef struct T1Choice /* CHOICE */
{
    enum T1ChoiceChoiceId
    {
        T1CHOICE_INT1,
        T1CHOICE_OID
    } choiceId;
    union T1ChoiceChoiceUnion
    {
        AsnInt int1; /* INTEGER */
        AsnOid* oid; /* OBJECT IDENTIFIER */
    } a;
} T1Choice;

typedef struct T1Seq /* SEQUENCE */
{
    AsnOid id; /* OBJECT IDENTIFIER */
    AsnOcts value; /* OCTET STRING */
} T1Seq;

typedef AsnList T1SeqOf; /* SEQUENCE OF INTEGER */

typedef struct T1 /* SEQUENCE */
{
    AsnInt* int1; /* INTEGER OPTIONAL */
    AsnOcts octs; /* OCTET STRING OPTIONAL */
    T1Enum t1Enum; /* T1Enum */
    T1SeqOf* t1SeqOf; /* T1SeqOf */
    struct T1Seq* t1Seq; /* T1Seq */
    struct T1Choice* t1Choice; /* T1Choice */
} T1;
```

Every ASN.1 type definition maps into a C `typedef`. SETs and SEQUENCEs map into C structures and other simple types map into their obvious C counterpart. SET OF and SEQUENCE OF types map into a generic list type which is doubly linked and NULL terminated. The reverse link on the lists allows for simpler backwards encoding. More information on the library types can be found in chapter 4.

Comments that contain a fragment of each type's ASN.1 definition are inserted in the header file to clarify cases where elements have been re-named.

Aggregate types that are defined in other type definitions are moved to their own type definitions. For example, notice how the SEQUENCE and CHOICE that are in type `T1` have been moved to the types `T1Seq` and `T1Choice` in the C code. This simplifies code generation at the cost of introducing new types.

Identifiers for named numbers from INTEGER and ENUMERATED types and named bits from the BIT STRING type are capitalized in the C representation. The ENUMERATED type maps to a C `enum` and the INTEGER and BIT STRING named numbers/bits are handled with `#define` statements.

Most OPTIONAL elements of SEQUENCEs and SETs are referenced by pointer. An element is considered present if its pointer is non-NULL. OCTET STRINGs, BIT STRINGs and OBJECT IDENTIFIERs are the exceptions, and are included by value even when they are OPTIONAL because they are small and contain an internal pointer that can be used to determine their presence. For an example of this, look at the first two elements of type `T1`. The INTEGER type is referenced by pointer because it is OPTIONAL, but the OCTET STRING type is included (non-pointer) in the `T1` type even though it is OPTIONAL.

3.4 Encode Routines

Snacc generates two kinds of encoding routines. One is PDU oriented and encodes the type's tag, length and content and the other only encodes the type's content. The generated encoders only call the content encoders, except in the case of ANY and ANY DEFINED BY types. Typically, you will only call the PDU oriented routines from your code.

The content and PDU encoding routine interfaces are similar for all ASN.1 types. They both take two parameters, one is a buffer pointer and the other is a pointer to the value to be encoded. For example the `T1` type from module EX1 has the following prototypes for its encoding routines.

```
AsnLen BEncT1Content(BUF_TYPE b, T1* v);
AsnLen BEncT1(BUF_TYPE b, T1* v);
```

`BEnc` is short for "BER Encode". The `BUF_TYPE` parameter is the buffer to encode the value into and the `T1*` parameter is a pointer to the instance of the `T1` type that is to be encoded.

The `BEncT1Content` routine only encodes the content of a `T1` type and returns its encoded length; it does not encode its tag (UNIVERSAL (CONSTRUCTED) 16 for SEQUENCE) or length. The job of encoding the tag and length is up to any type that encapsulates `T1`. This design allows decisions about implicit tagging to be made at code generation time instead of runtime, improving performance. Also, different encoding rules may fit into this model more easily.

The `BEncT1` routine encodes the tag (UNIVERSAL (CONSTRUCTED) 16 for SEQUENCE), length and content of a `T1` type and returns its encoded length. This is the PDU oriented routine and will only be generated if the user designates the type as a PDU type via a compiler directive or the type is used as the content of an ANY or ANY DEFINED BY type (as indicated by an OBJECT-TYPE macro). A PDU type is a type that defines an entire PDU; the user will typically be calling the encode and decode routine for PDU types directly. See Section 3.9 for how to designate PDU types with compiler directives.

The snacc encoders are somewhat strange; they encode a value starting from the end of its BER representation and work back to its beginning. This "backwards" encoding technique simplifies the use of definite lengths on constructed values. Other encoders that encode forwards, such as those of CASN1, use an intermediate buffer format so that a buffer containing the encoded length of a constructed value can be inserted before its encoded content, after the content has been encoded. Use of intermediate buffers hurts performance. Other compilers' approaches have been to only encode indefinite lengths for constructed values, however, this will not support some encoding rules such as DER. The drawback of encoding backwards is that BER values cannot be written to stream oriented connections as they are encoded.

Both definite and indefinite length encodings for constructed values' lengths are supported. Currently the choice is made when compiling the generated code, via the `USE_INDEF_LEN` flag. If both length forms, definite and indefinite, are required, it easy to modify the length encoding macros to to check a global variable for the length form to use. For most types, using definite lengths produces smaller encodings with little performance difference.

After calling and encode routine you should always check the buffer you encoded into for a write error. This is the only error reporting mechanism used for the encoders. See the C buffer section (Section 4.13) for how to check a buffer for a write error.

3.5 Decode Routines

Decoding routines are like the encoding routines in that there are two kinds, one that decodes the type's tag, length and content and one that only decodes the type's content. As mentioned in the encoder section, the content style interface allows implicit tagging decisions to be made at compile time.

Unlike the encoding routines, the PDU and content decoding routines take different arguments. For the `T1` type the following would be produced:

```
void BDecT1Content(BUF_TYPE b, AsnTag tagId0, AsnLen elmtLen0,
                   T1* v, AsnLen* bytesDecoded, ENV_TYPE env);
void BDecT1(BUF_TYPE b, T1* v, AsnLen* bytesDecoded, ENV_TYPE env);
```

Notice that the content decoder, `BDecT1Content`, has tag and length parameters that the PDU decoder, `BDecT1`, does not have. Since the content decoder does not decode the tag and length on the value, it is necessary to pass them in as parameters. Only OCTET STRING and BIT STRING decoders will use actually use the information in the tag parameter.

The `BUF_TYPE` parameter is the buffer that holds the BER value being decoded.

The `tagId0` parameter the last tag that was decoded on the content of the type that is about to be decoded. In the case of type `T1`, `BDecT1Content` gets a tagId0 of UNIVERSAL (CONSTRUCTED) 16, unless it is implicitly tagged by another type. Most content decoding routines ignore the tag information. OCTET STRING, and BIT STRING decoders use the tag information to determine whether the contents are constructed or primitive. CHOICE decoders use the tag information to determine which CHOICE element is present. CHOICE values are treated differently, as will be explained shortly.

The `elmtLen0` parameter is the length of the content of the type being decoded. This is simply the length decoded from the buffer by the containing type's decoder just before calling this decode routine.

The `v` parameter is a pointer to space allocated for the type being decoded. This memory is not allocated by the decoding routine itself; this supports the cases where the type is enclosed in the struct of the parent (i. e. no extra allocation is necessary). If the type to be decoded is referenced by pointer from its parent type, the parent type's decoding routine must allocate the type.

The `bytesDecoded` parameter maintains the running total of the number of octets that have been decoded. For example, if I call `BDecT1Content` with a `bytesDecoded` parameter that points to 20 and the encoded length of the `T1` value is 30 octets, `bytesDecoded` will point to 50 when `BDecT1Content` returns. Maintaining the length is vital to determining the presence or absence of OPTIONAL elements in a SET or at the end of SEQUENCE. Local variables are used to hold the lengths; there is no global stack of lengths as with CASN1.

The `env` parameter is used in conjunction with `longjmp` calls. When an decoder encounters a fatal error such as a missing tag, it uses the `env` with a `longjmp` call to pop back to the initial decode call. Section 4.15 has more error management details.

CHOICEs are decoded a little differently from other types. For all types except CHOICEs, all of the tag and length pairs on the content are decoded by the parent type, and the last pair a passed into to content decoding routine via the `tagId0` and `elmtLen0` parameters. For CHOICEs, all of the tag and length pairs on the content are decoded and then the first tag and length pair in the CHOICE content is decoded by the parent and passed into the CHOICE content decoding routine. The first tag in a CHOICE's content is the important tag by which the CHOICE determines which element is present. This technique simplifies the code for dealing with untagged CHOICEs embedded in other CHOICEs. CHOICEs nested in this way mean that a single tag determines which element is present in more than one CHOICE.

The decoding routines allocate memory to hold the decoded value. By default snacc decoders use nibble memory (see Section 4.14) which is very efficient in allocation and virtually cost free for freeing.

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

Snacc decoders can detect a variety of errors which will be reported by `longjmp`. Any tagging errors are reported. SETs must contain all non-OPTIONAL components and SEQUENCEs must be in order and contain all non-OPTIONAL components. Extra components in SETs and SEQUENCEs are considered an error. Errors will also be reported if you attempt to decode values that exceed the limitations of the internal representation (e. g. an integer that is larger than a `long int` allows).

3.6 Print Routines

All of the generated print routines take similar parameters. For example the `T1` type's print routine prototype is:

```
void PrintT1(FILE* f, T1* v, unsigned short int indent);
```

The print routine writes the given value, `v`, to the given FILE*, `f`. The printed value is indented by `indent` spaces. The values are printed in an ASN.1 value notation style. `PrintT1` prints the in the following style:

```
{ -- SEQUENCE --
    17,
    '436c696d622061742053717561616d697368'H  -- "Climb at Squamish" --,
    0,
    { -- SEQUENCE OF --
        18,
        19
    },
    { -- SEQUENCE --
        id {2 40 29},
        value '736f6d6520737472696e67'H  -- "some string" --
    },
```

```
    20
}
```

OCTET STRINGs are printed in a hexadecimal notation, and any printable characters are included after the string in an ASN.1 comment. Note that the enumerated type value, 0, did not print its symbol, "A" from the ENUMERATED type. It would be fairly easy to modify the C and C++ back ends to generate print routines that printed the ENUMERATED types' symbols instead of their values.

3.7 Free Routines

Snacc generates free routines of the form:

```
void FreeT1(T1* v);
```

These routines will free all the components named type. For example the above `FreeT1` routine will free all the components of the given `T1` value, but not the `T1` value itself. The passed in pointer is not freed because it may be embedded in another type which will be freed by another call to `Asn1Free`. All the pieces of memory are freed using the `Asn1Free` macro defined in `asn_config.h`. Each library type has its own free routine that may call `Asn1Free`. The values are typically allocated during decoding, using the `Asn1Alloc` macro.

The memory management can be changed by editing the `asn_config.h` file to use you own memory management routines. By default the memory manager uses the nibble memory system described in Section 4.14. The nibble memory system does not need explicit frees of each component so the generated free routines are not needed. However, if you change the memory management to use something like `malloc` and `free`, you should use the generated free routines.

3.8 ASN.1 to C Value Translation

C values will be produced for INTEGER, BOOLEAN and OBJECT IDENTIFIER values. C `extern` declarations for the value are put at the end of the header file (after all of the type definitions). The value definitions are put at the beginning of the source file. For example, the following will be produced for the EX1 module.

at the end of file ex1.h:

```
extern AsnOid anOidVal;
extern AsnOid theSameOidVal;
extern AsnInt anIntVal;
extern AsnBool aBoolVal;
extern AsnInt foobar;
```

at the beginning of file ex1.c:

```
AsnOid anOidVal = { 2, "\170\35" };
AsnOid theSameOidVal = { 2, "\170\35" };
AsnInt anIntVal = 1;
AsnBool aBoolVal = TRUE;
AsnInt foobar = 29;
```

3.9 Compiler Directives

Snacc allows the user to control some aspects of the generated code by inserting special comments in the ASN.1 source. Warning! only the `isPdu` directive has been tested to any extent. Use the others very carefully and only if you really need to. The compiler directives have the form:

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

The `attribute` is the name of one of the accepted attributes the `value` is what the `attribute`'s new value will be. The attribute value pairs can be listed in a single `--snacc` comment or spread out is a list of consecutive comments.

Compiler directives are only accepted in certain places in the ASN.1 code. Depending on their location in the ASN.1 source, the compiler directives affect type definitions or type references. The directives for type definitions and references are different. Module level compiler directives to specify output file names and other information would be useful, but are not implemented.

Here is an example to present some of the compiler directives and their uses. Lets say your data structure always deals with `PrintableStrings` that are null terminated (internally, not in the encoding). The default snacc string type is a structure that includes a length and `char*` for the string octets. To change the default type to a simple `char*` the best way would be define your own string type, lets say `MyString` as follows:

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

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

```
Bell ::= MyString

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

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

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

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

typedef struct Bar /* CHOICE */
{
    enum BarChoiceId
    {
        BAR_S1,
        BAR_I1
    } choiceId;
    union BarChoiceUnion
    {
    MyString s1; /* MyString */
    AsnInt i1; /* INTEGER */
    } a;
} Bar;

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

The compiler directives used on the `MyString` type have some interesting effects. Notice that `MyString` is not referenced by pointer in the CHOICE, SET, or type definition, `Bell`.

The generated code for encoding field `s1` of `Foo` type will use the code "`MYSTRING_NON_NULL(&fooVal->s1)`" to check for the presence of the OPTIONAL `s1` field. The code associated with MYSTRING_NON_NULL should return TRUE if the `s1` field value is present and might look like:

```
#define MYSTRING_NON_NULL(s) (*s != NULL)
```

The argument to `optTestRoutine` routine will be a pointer to the field type's defining type. Note that in the above example, `MyString` is a `char*`, therefore the `MYSTRING_NON_NULL` macro's argument will be a `char**`.

Setting the `genPrintRoutine` etc. attributes to false makes snacc not define or generate any encode, decode, print, or free routines for the `MyString` type. You must provide these yourself; the best approach is to take the normal `PrintableString` routines and modify them to handle your special string type.

The names of the encode, decode, print and free routines used for the `MyString` type will be based on the ones given with the `printRoutineName` etc. attributes. Snacc will prepend and a "B" (for BER) and append a "Content" to the encode and decode routines names, so you must provide the `BEncMyStringContent` and `BDecMyStringContent` routines. You may also need the `BEncMyString` and `BDecMyString` routines if `MyString` is a PDU type or used in an ANY or ANY DEFINED type.

The `PrintableString` type has its C type name changed to `char*` by the last compiler directive. Thus `MyString` is defined as a `char*`. This directive applies to the `PrintableString` type reference. Note that these directives do not affect the tags or the encoded representation of the `MyString` type

The location of the `--snacc` comment(s) is important. `--snacc` comment(s) between the `::=` sign and the following type are associated with the type being defined. Any compiler directives after the type and before the next type or value definition are associated with the type. Fields in SETs, SEQUENCEs and CHOICEs can be modified by putting the compiler directive after the comma that follows the field type that you with to modify. In the case of the last element of one of these types, where there is no comma, just place it after the field and before the closing bracket of the parent type.

Attributes shadow the type attributes filled in during the target language type information generation pass of the compiler. The type definition attributes are:

cTypeName this is the type name that the generated type will have. Its value can be any string that is valid as a C type name.

isPdu whether this is a PDU type. A PDU type will have extra interfaces to the encode and decode routines generated. Its value can be "TRUE" or "FALSE"

isPtrForTypeDef TRUE if other types defined solely by this type definition are defined as a pointer to this type. Its value can be "TRUE" or "FALSE".

isPtrForTypeRef TRUE if type references to this type definition from a SET or SEQUENCE are by pointer. Its value can be "TRUE" or "FALSE".

isPtrInChoice TRUE if type references to this type definition from a CHOICE are by pointer. Its value can be "TRUE" or "FALSE".

isPtrForOpt TRUE if OPTIONAL type references to this type definition from a SET or SEQUENCE are by pointer. Its value can be "TRUE" or "FALSE".

optTestRoutineName name of the routine to test whether an OPTIONAL element of this type in a SET or SEQUENCE is present. The routine should return TRUE if the element is present. The value of this field is usually just the name of a C macro that tests for NON-NULL. The argument to the routine will be a pointer to the type definition's type. The optTestRoutineName value can be any string value.

defaultFieldName if a this type is used in a SET, SEQUENCE or CHOICE without a field name then this value is used with a digit appended to it. Its value can be any string that is a valid C field name in a struct or union.

printRoutineName name of this type definition's printing routine. Its value can be any string that is a value C function or macro name.

encodeRoutineName name of this type definition's encoding routine. Its value can be any string that is a value C function or macro name.

decodeRoutineName name of this type definition's decoding routine. Its value can be any string that is a value C function or macro name.

freeRoutineName name of this type definition's freeing routine. Its value can be any string that is a value C function or macro name.

isEncDec If this type is used in a SET or SEQUENCE then it is not encoded or decoded. Its value can be "TRUE" or "FALSE". This is handy for adding your own types to a standard that are only for local use, and are not included in encoded values.

genTypeDef TRUE if you want a C type to be generated for this type definition. Its values can be "TRUE" or "FALSE".

genPrintRoutine TRUE if you want a printing routine to be generated for this type definition. Its values can be "TRUE" or "FALSE".

genEncodeRoutine TRUE if you want an encoding routine to be generated for this type definition. Its values can be "TRUE" or "FALSE".

genDecodeRoutine TRUE if you want a decoding routine to be generated for this type definition. Its values can be "TRUE" or "FALSE".

genFreeRoutine TRUE if you want a free routine to be generated for this type definition. Its values can be "TRUE" or "FALSE".

The type reference attributes are slightly different from the type definition attributes due to the semantic differences between a type definition and a type reference. Type references will inherit some of their attributes from the referenced type definition. The following are the valid type reference attributes:

cTypeName this is the type name that the generated type will have. Its value can be any string that is valid as a C type name.

cFieldName if this is a field in a CHOICE, SET or SEQUENCE then this holds the C field name for this reference. Its value can be any string that is valid as a C field name.

isPtr TRUE if this is a pointer to the type named by cTypeName. This is usually determined from the referenced type definitions attributes. Its value can be "TRUE" or "FALSE".

optTestRoutineName if this field is an OPTIONAL component then this is the name of the routine to test whether it is present. The routine should return TRUE if the element is present. The value of Usually just the name of a C macro that tests for NULL. The argument to the routine will be a pointer to the type definition's type. The optTestRoutineName value can be any string value.

printRoutineName name of this type reference's printing routine. This and the other routine name attributes are useful for special instances of the referenced type. It is easier to modify the referenced type definition if you want every instance of this type to use a certain print etc. routine. Its value can be any string that is a value C function or macro name.

encodeRoutineName name of this type reference's encoding routine. Its value can be any string that is a value C function or macro name.

decodeRoutineName name of this type reference's decoding routine. Its value can be any string that is a value C function or macro name.

freeRoutineName name of this type reference's freeing routine. Its value can be any string that is a value C function or macro name.

isEncDec If this type is used in a SET or SEQUENCE then it is not encoded or decoded. Its value can be "TRUE" or "FALSE". This is handy for adding your own types to a standard that are only for local use, and are not inlcuded in encoded values.

choiceIdSymbol if this is a component of a CHOICE, this string attribute will be the defined/enum symbol whose value in the choiceId field indicates the presence of this field.

choiceIdValue if this is a component of a CHOICE, this integer attribute will be the value associated with the symbol in choiceIdSymbol.

3.10 Compiling the Generated C Code

The generated C code (and libraries) can be compiled by both ANSI and non-ANSI C compilers. C function prototypes use the `PROTO` macro and C function declarations use the `PARAMS` macro. These macros are defined in `snacc/include/asn_config.h` and their definitions depend on whether the `__USE_ANSI_C__` flag has been defined. When compiling the generated code you will need:

1 `snacc/c_include` directory in your include path (`-I snacc/c_include`)

2 to link with one of the C ASN.1 runtime libraries in `snacc/c_lib`. There are three libraries to choose from, their only difference is the buffer types they use.

3 to link with the math library, since the ASN.1 REAL type's encode and decode routine use some math routines. (`-lm`)

See the example in `snacc/c_examples/simple` for a complete example. The makefile and main routines are probably the most important. There are several other examples in the `snacc/c_examples` directory.

Chapter 4

C ASN.1 Library

4.1 Overview

Each library type has a file in the `snacc/c_lib` and `snacc/c_include` directories. Each source file contains the encode, decode, free and print routines for the given type. This chapter contains a description of each library type and its routines. This library is also refered to as the runtime library.

After installing snacc, you should test the library types to make sure that they are encoding and decoding properly. Use the `snacc/c_example/test_lib` example to check them.

In addition to other errors, most decoding routines will report an error if they attempt to read past the end of the data. Be aware that some buffer types do not support this type of checking. This is explained more in the buffer management section.

4.2 Tags

Snacc's tag representation was motivated by several things.

1 the tags must be easy to compare for equality in `if` and `switch` statements to make tag based decisions cheap.

2 a tag must be cheap to decode.

3 a tag must be cheap to encode.

The first requirement meant that tags had to be integer types (for the `switch statement`). The representation of the tag within the integer was set by the second requirement.

The best way to decode cheaply is minimize the transformation between the encoded and decoded (internal) format. So the four (can be set-up for two) bytes of the long integer are used to hold the encoded tag, starting with the first octet of the tag in the most significant byte of the integer and the rest (if any) following. Any unused (always trailing) bytes in the integer are zero. This limits the representable tag code to less than 2^{21} but for reasonable ASN.1 specifications this should not be a problem.

To meet the third requirement the decoded tag representation was bypassed entirely by using macros (`BEncTag1()` etc.) that write the encoded tag octet(s) to the buffer. The writing of an encoded tag octet involves bit shifting, bitwise ands and bitwise ors with constant values; most optimizing C compilers can compute these at compile time. This simplifies encoding a tag to writing some constant byte value(s) to the buffer.

The following excerpt from `snacc/c_include/asn_tag.h` shows some of the tag routines.

```
typedef unsigned long int AsnTag;

#define MAKE_TAG_ID( class, form, code) ...
#define TAG_IS_CONS(tag) ...

#define BEncTag1( b, class, form, code) ...
#define BEncTag2( b, class, form, code) ...
#define BEncTag3( b, class, form, code) ...
#define BEncTag4( b, class, form, code) ...
#define BEncTag5( b, class, form, code) ...

AsnTag BDecTag(BUF_TYPE b, AsnLen* bytesDecoded, ENV_TYPE env);
```

The generated decode routines use the `BDecTag` to decode a tag from the buffer. The returned tag value is either used in an `if` expression or as the argument to `switch` statements. The `MAKE_TAG_ID` macro is used to make a tag for comparison to the one returned by `BDecTag`. The `MAKE_TAG_ID` is used `switch` statement case labels and in `if` statements.

Most of the time tags are only compared for equality, however, the OCTET STRING and BIT STRING decoders check the constructed bit in the tag using the `TAG_IS_CONS` macro.

The `BEncTag` macros are quite fragile because they return the encoded length of the tag; they cannot be treated as a single statement. This requires careful use of braces when using them in your own code in places such as the sole statement in an `if` statement. This ugliness is caused by the difficulty in returning values from multi-line macros (macros are used for performance here since encoding tags can be a significant part of BER encoding).

The `BDecTag` routine will report an error via `longjmp` if the encoded tag is longer than can be held in the `AsnTag` type or if the it read past the end of the data when decoding the tag.

4.3 Lengths

Decoded lengths are represented by unsigned long integers, with the maximum value indicating indefinite length.

Snacc users can choose between using only indefinite or only definite lengths when encoding constructed values' lengths when compiling the generated code. Of course, the generated decoders can handle both forms. Define the `USE_INDEF_LEN` symbol when compiling the generated code if you want to use indefinite lengths when encoding constructed values. Primitive values are always encoded with definite lengths as required by the standard; this is necessary to avoid confusion between a value's content and the End-Of-Contents marker.

There is no loss of performance when using definite lengths with snacc encoders. This is due the "backwards" encoding as described in section 3.4. The schemes used by other compilers' encoders to handle definite lengths may hurt performance.

Most of the routines in the following code are obvious except for `BEncDefLenTo127()`. This is used instead of `BEncDefLen` in the generated code when the compiler knows the value being encode will not be over 127 octets long. Values such as BOOLEANS, INTEGERs, and REALs are assumed to be shorter than 127 octets (constraints on the decoded representation of INTEGERs and REALs make this valid).

```
typedef unsigned long int AsnLen;

/* max unsigned value  - used for internal rep of indef len */
#define INDEFINITE_LEN  ~0L

#ifdef USE_INDEF_LEN
#define BEncEocIfNec(b) BEncEoc(b);
#define BEncConsLen(b, len) 2 + BEncIndefLen(b)
#else
#define BEncEocIfNec(b)
#define BEncConsLen(b, len) BEncDefLen(b, len)
#endif

#define BEncIndefLen(b) ...
#define BEncDefLenTo127(b, len) ...
AsnLen  BEncDefLen(BUF_TYPE b, AsnLen len);
AsnLen  BDecLen(BUF_TYPE b, AsnLen* bytesDecoded, ENV_TYPE env);

#define BEncEoc(b) ...
#define BDEC_2ND_EOC_OCTET(b, bytesDecoded, env) ...
void    BDecEoc(BUF_TYPE b, AsnLen* bytesDecoded, ENV_TYPE env);
```

The `BDecLen` routine will report an error via `longjmp` if it attempts to read past the end of the data or the decoded length is too large to be held in the `AsnLen` representation. `BDecEoc` will report an error if it attempts to read past the end of the data or one of the EOC (End-Of-Contents) octets is non-zero.

4.4 BOOLEAN

The BOOLEAN type is represented by an `unsigned char`. It has the following routines for manipulating it.

```
typedef unsigned char AsnBool;

AsnLen BEncAsnBool(BUF_TYPE b, AsnBool* data);
void   BDecAsnBool(BUF_TYPE b, AsnBool* result, AsnLen* bytesDecoded,
                   ENV_TYPE env);

AsnLen BEncAsnBoolContent(BUF_TYPE b, AsnBool* data);
void   BDecAsnBoolContent(BUF_TYPE b, AsnTag tag, AsnLen len,
                          AsnBool* result, AsnLen* bytesDecoded,
                          ENV_TYPE env);

#define FreeAsnBool(v)
void PrintAsnBool(FILE* f, AsnBool* b, unsigned short int indent);
```

As discussed in Sections 3.4 and 3.5, `BEncAsnBool` and `BDecAsnBool` encode/decode the UNIVERSAL tag, length and content of the given BOOLEAN value. The `BEncAsnBoolContent` and `BDecAsnBoolContent` routine only encode/decode the content of the given BOOLEAN value.

The `FreeAsnBool` routine does nothing since the BOOLEAN type does not contain pointers to data; The free routine generator does not have to check which types need freeing and simply calls the type's free routine. It also allows the user to modify the types and their free routines without changing the free routine generator. However, the ANY and ANY DEFINED BY type hash table init routine generator does need to know which types have empty free routines because the hash entries contain pointers to the free functions (NULL is used for the empty free functions like `FreeAsnBool`). The INTEGER, NULL, REAL and ENUMERATED types have empty free routines for the same reason.

`BDecAsnBool` will report an error if the tag is not UNIVERSAL-PRIM-1. `BDecAsnBoolContent` will report an error if it decodes past the end of the data or the length of the encoded value (given by the `len` parameter) is not exactly 1 octet.

4.5 INTEGER

The INTEGER type is represented by a long int, `AsnInt`. If you are using INTEGER types that are only positive (via subtyping or protocol definition) you may want to use the `UAsnInt` and associated routines that use the unsigned long int for a larger positive value range.

```
typedef long int AsnInt;
typedef unsigned long int UAsnInt;

AsnLen BEncAsnInt(BUF_TYPE b, AsnInt* data);
void BDecAsnInt(BUF_TYPE b, AsnInt* result, AsnLen* bytesDecoded,
                ENV_TYPE env);

AsnLen BEncAsnIntContent(BUF_TYPE b, AsnInt* data);
```

```
void BDecAsnIntContent(BUF_TYPE b, AsnTag tag, AsnLen elmtLen,
                       AsnInt*  result, AsnLen* bytesDecoded,
                       ENV_TYPE env);

#define FreeAsnInt(v)
void PrintAsnInt(FILE* f, AsnInt* v, unsigned short int indent);

AsnLen BEncUAsnInt(BUF_TYPE b, UAsnInt* data);
void BDecUAsnInt(BUF_TYPE b, UAsnInt* result, AsnLen* bytesDecoded,
                 ENV_TYPE env);

AsnLen BEncUAsnIntContent(BUF_TYPE b, UAsnInt* data);
void BDecUAsnIntContent(BUF_TYPE b, AsnTag tagId, AsnLen len,
                        UAsnInt* result, AsnLen* bytesDecoded,
                        ENV_TYPE env);

#define FreeUAsnInt(v)
void PrintUAsnInt(FILE* f, UAsnInt* v, unsigned short int indent);
```

`BDecAsnInt` will report an error if the tag is not UNIVERSAL-PRIM-2. `BDecAsnIntContent` will report an error if it decodes past the end of the data or the integer value is too large for an `AsnInt`.

4.6 NULL

The NULL type is represented by the `AsnNull` type. Its content is always empty and hence its encoded length always is zero.

```
typedef char AsnNull;

AsnLen BEncAsnNull(BUF_TYPE b, AsnNull* data);
void BDecAsnNull(BUF_TYPE b, AsnNull* result, AsnLen* bytesDecoded,
                 ENV_TYPE env);

/* 'return' length of encoded NULL value, 0 */
#define BEncAsnNullContent(b, data) 0
void BDecAsnNullContent(BUF_TYPE b, AsnTag tag, AsnLen len,
                        AsnNull* result, AsnLen* bytesDecoded,
                        ENV_TYPE env);

#define FreeAsnNull(v)
void PrintAsnNull(FILE* f, AsnNull * b, unsigned short int indent);
```

4.7 REAL

The REAL type is represented by `AsnReal`, a double. This type's representation can depend on the compiler or system you are using so several different encoding routines are provided. Even so, you may need to modify the code.

If you are using the REAL type in your ASN.1 modules, you should call the `InitAsnInfinity()` routine to setup the `PLUS_INFINITY` and `MINUS_INFINITY` values.

There are three encode routines included and they can be selected by defining one of `IEEE_REAL_FMT`, `IEEE_REAL_LIB` or nothing. Defining `IEEE_REAL_FMT` uses the encode routine that assumes the double representation is the standard IEEE double [3]. Defining `IEEE_REAL_LIB` uses the encode routine that assumes the IEEE functions library (isinf, scalbn, signbit etc.) is available. If neither are defined, the default encode routine uses `frexp`.

There is only one content decoding routine and it builds the value through multiplication and the `pow` routine (requires the math library). The content decoding routine only supports the binary encoding a of REAL, not the decimal encoding.

```
typedef double AsnReal;

extern AsnReal PLUS_INFINITY;
extern AsnReal MINUS_INFINITY;

void InitAsnInfinity();
AsnLen BEncAsnReal(BUF_TYPE b, AsnReal* data);
void BDecAsnReal(BUF_TYPE b, AsnReal* result, AsnLen* bytesDecoded,
                       ENV_TYPE env);

AsnLen BEncAsnRealContent(BUF_TYPE b, AsnReal* data);
void BDecAsnRealContent(BUF_TYPE b, AsnTag tag, AsnLen len,
                               AsnReal* result, AsnLen* bytesDecoded,
                               ENV_TYPE env);

/* do nothing */
#define FreeAsnReal(v)
void PrintAsnReal(FILE* f, AsnReal* b, unsigned short int indent);
```

`BDecAsnReal` will report an error if the value's tag is not UNIVERSAL-PRIM-9. `BDecAsnRealContent` will report an error if the base is not supported or the decimal type REAL encoding is received.

4.8 BIT STRING

The BIT STRING type is represented by the `AsnBits` structure. It contains a pointer to the bits and and integer that holds the length in bits of the BIT STRING.

In addition to the standard encode, decode, print and free routines, there are some other utility routines. `AsnBitsEquiv` returns TRUE if the given BIT STRINGs are identical. The `SetAsnBit`, `ClrAsnBit` and `GetAsnBit` are routines for writing and reading a BIT STRING value.

You may notice that the AsnBits type does not have any means of handling linked pieces of BIT STRINGs. Some ASN.1 tools use lists of structures like `AsnBits` to represent BIT STRINGs. This is done because, as you should be aware, BIT STRINGs can be encoded in a nested, constructed fashion. The snacc BIT STRING decoder attempts to save you the hassle of dealing with fragments of BIT STRINGs by concatenating them in the decoding step. Every BIT STRING value returned by the decoder will have contiguous bits.

Some people contend that fragmented BIT STRINGs are necessary to support systems that lack enough memory to hold the entire value. Snacc encodes value "backwards" so the entire value must be encoded before it can be sent, thus you must have enough memory to hold the whole encoded value. If the fragmented representation is useful to your protocol implementation for other reasons, it should be fairly simple to modify the BIT STRING routines. Remember, no significance should be placed on where constructed BIT STRING values are fragmented.

Snacc uses a table to hold pointers to the BIT STRING fragments in the buffer while it is decoding them. Once the whole BIT STRING value has been decoded, a block of memory that is large enough to hold the entire BIT STRING is allocated and the fragments are copied into it. The table initially can hold pointers to 128 fragments. If more table entries are needed the stack will grow via `realloc` (with associated performances loss) and will not shrink after growing. If you wish to modify this behaviour, change the `snacc/c_include/str_stk.h` file.

The `FreeAsnBits` routine will free memory referenced by the `bits` pointer.

```
typedef struct AsnBits
{
    int   bitLen;
    char* bits;
} AsnBits;

extern char numToHexCharTblG[];
#define TO_HEX(fourBits) (numToHexCharTblG[(fourBits) & 0x0f])
#define ASNBITS_PRESENT(abits) ((abits)->bits != NULL)

AsnLen BEncAsnBits(BUF_TYPE b, AsnBits* data);
void BDecAsnBits(BUF_TYPE b, AsnBits* result, AsnLen* bytesDecoded,
                 ENV_TYPE env);

AsnLen BEncAsnBitsContent (BUF_TYPE b, AsnBits* bits);
void BDecAsnBitsContent(BUF_TYPE b, AsnLen len, AsnTag tagId,
                        AsnBits* result, AsnLen* bytesDecoded,
                        ENV_TYPE env);
```

```
void FreeAsnBits(AsnBits* v);
void PrintAsnBits(FILE* f, AsnBits* b, unsigned short int indent);

int AsnBitsEquiv(AsnBits* b1, AsnBits* b2);
void SetAsnBit(AsnBits* b1, unsigned long int bit);
void ClrAsnBit(AsnBits* b1, unsigned long int bit);
int GetAsnBit(AsnBits* b1, unsigned long int bit);
```

`BDecAsnBits` will report an error if the tag is not UNIVERSAL-CONS-3 or UNIVERSAL-PRIM-3. When decoding constructed BIT STRING BER values, an error will be reported if a component other than the last one has non-zero unused bits in its last octet or an internal component does not have the UNIVERSAL-3 tag. If the decoder attempts to read past the end of the data an error will be reported.

4.9 OCTET STRING

The OCTET STRING type is represented by the `AsnOcts` structure. It contains a pointer to the octets and an integer that holds the length in octets of the OCTET STRING.

As with BIT STRINGs, OCTET STRINGs can have constructed values. These are handled in the same way as the constructed BIT STRING values. The decoded representation of an OCTET STRING is always contiguous.

The `FreeAsnOcts` routine will free the memory referenced by the `octs` pointer. The `AsnOctsEquiv` routine will return TRUE if the given OCTET STRINGs are identical.

```
typedef struct AsnOcts
{
    unsigned long int octetLen;
    char* octs;
} AsnOcts;

#define ASNOCTS_PRESENT(aocts) ((aocts)->octs != NULL)

AsnLen BEncAsnOcts(BUF_TYPE b, AsnOcts* data);

void BDecAsnOcts(BUF_TYPE b, AsnOcts* result, AsnLen* bytesDecoded,
                 ENV_TYPE env);

AsnLen BEncAsnOctsContent (BUF_TYPE b, AsnOcts* octs);
void BDecAsnOctsContent (BUF_TYPE b, AsnLen len, AsnTag tagId,
                         AsnOcts* result, AsnLen* bytesDecoded,
                         ENV_TYPE env);

void FreeAsnOcts (AsnOcts* o);
void PrintAsnOcts (FILE* f, AsnOcts *o, unsigned short int indent);
```

```
int AsnOctsEquiv (AsnOcts* o1, AsnOcts* o2);
```

BDecAsnOcts will report an error if the tag is not UNIVERSAL-CONS-4 or UNIVERSAL-PRIM-4. When decoding constructed OCTET STRING BER values, an error will be reported if an internal component does not have the UNIVERSAL-4 tag. if the decoder attempts to read past the end of the data an error will be reported.

4.10 OBJECT IDENTIFIER

In snacc, OBJECT IDENTIFIERs are kept in their encoded form to improve performance. The AsnOid type is defined as AsnOcts, as it holds the octets of the encoded OBJECT IDENTIFIER. It seems that the most common operation with OBJECT IDENTIFIERs is to compare for equality, for which the encoded representation (which is canonical) works well.

There is a linked OBJECT IDENTIFIER representation called OID and routines to convert it to and from the AsnOid format, but it should not be used if performance is an issue.

Since the OBJECT IDENTIFIERs are represented AsnOcts, the AsnOcts content encoding routine can be used for the AsnOid content encoding routine. The other AsnOcts encoding and decoding routines cannot be used because the OBJECT IDENTIFIER has a different tag and cannot be encoded in a constructed fashion.

An OBJECT IDENTIFIER must have a minimum of two arc numbers but the decoding routines do not check this.

```
typedef AsnOcts AsnOid;

#define ASNOID_PRESENT(aoid) ASNOCTS_PRESENT(aoid)

AsnLen BEncAsnOid(BUF_TYPE b, AsnOid* data);
void BDecAsnOid(BUF_TYPE b, AsnOid* result, AsnLen* bytesDecoded,
                ENV_TYPE env);

#define BEncAsnOidContent(b, oid)  BEncAsnOctsContent(b, oid)
void BDecAsnOidContent(BUF_TYPE b, AsnTag tag, AsnLen len,
                       AsnOid*  result, AsnLen* bytesDecoded,
                       ENV_TYPE env);

#define FreeAsnOid FreeAsnOcts
void PrintAsnOid (FILE* f, AsnOid* b, unsigned short int indent);

#define AsnOidsEquiv(o1, o2) AsnOctsEquiv(o1, o2)
```

4.11 SET OF and SEQUENCE OF

The SET OF and SEQUENCE OF type are represented by the `AsnList` structure. An `AsnList` consists of a head object that has pointers to the first, current and last nodes and the current number of nodes in the list. Each list node has a pointer to its next and previous list member and the node's data. The first list node's previous pointer is always NULL and the last list node's next pointer is always NULL.

Each SET OF or SEQUENCE OF type is defined as an `AsnList`, so the element type information (kept via a `void*`) is not kept, therefore, the `AsnList` type is not type safe.

The `AsnList` is doubly linked lists to simplify "backwards" encoding. The reverse link allows the list to be traversed in reverse so the components can be encoded from last to first.

Initially, the lists were designed to allow the list element itself to be contained in the list node (hence the `elmtSize` parameter to the AsnListNew() routine). The design eventually changed such that every list element was reference by pointer from the list node.

As small problem with the `AsnListNew` routine is the memory allocation. Since it is used by the decoding routines to allocate new lists, it uses whatever memory management you have setup with the `Asn1Alloc` macro (see Section 4.14). This may not be desirable when building values to be transmitted. You may need to provide another AsnListNew routine that uses a different allocation scheme to solve this.

```
typedef struct AsnListNode
{
    struct AsnListNode *prev;
    struct AsnListNode *next;
    void            *data; /* this must be the last field of this structure  */
} AsnListNode;

typedef struct AsnList
{
    AsnListNode *first;
    AsnListNode *last;
    AsnListNode *curr;
    int        count;    /* number of elements in list               */
    int        dataSize; /* space required in each node for the data */
} AsnList;

#define  FOR_EACH_LIST_ELMT( elmt, list) ...
#define  FOR_EACH_LIST_ELMT_RVS( elmt, list) ...
#define  FOR_REST_LIST_ELMT( elmt, al) ...

#define CURR_LIST_ELMT(al)    (al)->curr->data
#define NEXT_LIST_ELMT(al)    (al)->curr->next->data
#define PREV_LIST_ELMT(al)    (al)->curr->prev->data
#define LAST_LIST_ELMT(al)    (al)->last->data
```

```
#define FIRST_LIST_ELMT(al)   (al)->first->data
#define LIST_EMPTY(al) ((al)->count == 0)

#define CURR_LIST_NODE(al) ((al)->curr)
#define FIRST_LIST_NODE(al) ((al)->first)
#define LAST_LIST_NODE(al) ((al)->last)
#define PREV_LIST_NODE(al) ((al)->curr->prev)
#define NEXT_LIST_NODE(al) ((al)->curr->next)
#define SET_CURR_LIST_NODE(al, listNode)  ((al)->curr = (listNode))

void  AsnListRemove(AsnList* l);
void* AsnListAdd(AsnList* l);
void* AsnListInsert(AsnList* list);
void  AsnListInit(AsnList* list, int dataSize);
AsnList* AsnListNew(int elmtSize);
void* AsnListPrev(AsnList*);
void* AsnListNext(AsnList*);
void* AsnListLast(AsnList*);
void* AsnListFirst(AsnList*);
void* AsnListPrepend(AsnList*);
void* AsnListAppend(AsnList*);
void* AsnListCurr(AsnList*);
int   AsnListCount(AsnList*);
AsnList* AsnListConcat(AsnList*, AsnList*);
```

There are a number of macros for dealing with the list routine, the most important being the list traversal macros. The `FOR_EACH_LIST_ELMT` macro acts like a for statment that traverses forward through the list. The first parameter should be a pointer to the list element type that will be used to hold the current list element for each iteration of the for loop. The second parameter is the list of elements that you wish to traverse.

The `FOR_EACH_LIST_ELMT_RVS` macro is identical to the `FOR_EACH_LIST_ELMT` macro except that is moves from the back of the list to the front. The `FOR_REST_LIST_ELMT` macro is similar to the other two but it does not reset the `curr` pointer in the `AsnList` type. This has the effect of iterating from the current element to the end of the list. Look in the generated code for a better indication of how to use these macros. The other macros are straight forward.

4.12 ANY and ANY DEFINED BY

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

The ANY DEFINED BY type can be handled automatically by snacc provided you use the SNMP OBJECT-TYPE macro to specify the identifier to type mapping. The identifier can be an INTEGER or OBJECT IDENTIFIER. Handling ANY types properly will require modifications to the generated code since there is no identifier associated with the type.

Look at the C and C++ ANY examples and `any.asn1` included with this release for information on using the OBJECT-TYPE macro. Note that the OBJECT-TYPE macro has been modified slightly to allow INTEGER values (identifiers).

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

Two hash tables are used, one for INTEGER to type mappings and the other for OBJECT IDENTIFIER to type mappings. Snacc generates an `InitAny`*ModuleName* routine to add mappings to the hash table(s) for each module that uses the OBJECT-TYPE macro to define identifier to type mappings. For example with a module named `"ANY-TEST"`, snacc will generate an `InitAnyANY_TEST()` routine. The `InitAny` routine(s) should be called only once, before any encoding or decoding is done. If there is more than one `InitAny` routine, the order in which they are called is not relevant.

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

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

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

Snacc builds an `enum` called *ModuleName*`AnyId` that enumerates each mapping defined by the OBJECT-TYPE macros. The name of the value associated with each macro is used as part of the enumerated identifier. The `anyId` in the hash table holds the identified type's enum value. The `anyId` is handy for making decisions based on the received identifier, without comparing OBJECT IDENTIFIERs. If the identifiers are INTEGERs then the `anyId` is less useful.

With ANY DEFINED BY types, it is important to have the identifier decoded before the ANY DEFINED BY type is decoded. Hence, you should not put an ANY DEFINED BY type and identifier in a SET since they are un-ordered and you definitely should not put an

ANY DEFINED BY type before its identifier in a SEQUENCE. Snacc will print a warning if any of these situations occur.

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

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

To illustrate how things work, here is an brief outline of what happens when encoding and decoding. Each ANY or ANY DEFINED BY type is represented in C by the `AsnAny` type which contains only a `void* value` to hold a pointer to the value and `AnyInfo* ai` which points to a hash table entry (as described above).

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

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

Here is some of the `AsnAny` related code from the header file. It should help you understand the way things are done a bit better. Look in the `hash.c` and `hash.h` files as well.

```
/*
 * 1 hash table for integer keys
 * 1 hash table for oid keys
 */
extern Table* anyOidHashTblG;
```

```
extern Table* anyIntHashTblG;

typedef (*EncodeFcn) (BUF_TYPE b, void* value);
typedef void (*DecodeFcn) (BUF_TYPE b, void* value,
                           AsnLen* bytesDecoded, ENV_TYPE env);
typedef void (*FreeFcn)   (void* v);
typedef void (*PrintFcn)  (FILE* f, void* v);

/*
 * this is put into the hash table with the
 * int or oid as the key
 */
typedef struct AnyInfo
{
    int    anyId;  /* will be a value from the AnyId enum */
    AsnOid oid;    /* will be zero len/null if intId is valid */
    AsnInt intId;
    unsigned int size;  /* size of the C data type (ie as ret'd by sizeof) */
    EncodeFcn Encode;
    DecodeFcn Decode;
    FreeFcn Free;
    PrintFcn Print;
} AnyInfo;

typedef struct AsnAny
{
    AnyInfo* ai; /* point to entry in hash tbl that has routine ptrs */
    void* value; /* points to the value */
} AsnAny;

/*
 * Returns anyId value for the given ANY type.
 * Use this to determine to the type of an ANY after decoding
 * it. Returns -1 if the ANY info is not available
 */
#define GetAsnAnyId(a) (((a)->ai)? (a)->ai->anyId: -1)

/*
 * used before encoding or decoding a type so the proper
 * encode or decode routine is used.
 */
void SetAnyTypeByInt (AsnAny* v, AsnInt id);
void SetAnyTypeByOid (AsnAny* v, AsnOid* id);


/*
 * used to initialize the hash table(s)
 */
void InstallAnyByInt(int anyId,  AsnInt intId,
                     unsigned int size, EncodeFcn encode,
                     DecodeFcn decode, FreeFcn free, PrintFcn print);
```

```
void InstallAnyByOid(int anyId, AsnOid* oid, unsigned int size,
                     EncodeFcn encode, DecodeFcn decode, FreeFcn free,
                     PrintFcn print);

/*
 * Standard enc, dec, free, & print routines.
 * for the AsnAny type.
 * These call the routines referenced from the
 * given value's hash table entry.
 */
void FreeAsnAny (AsnAny* v);
AsnLen BEncAsnAny (BUF_TYPE b, AsnAny* v);
void BerDecAsnAny(BUF_TYPE b, AsnAny*  result, AsnLen* bytesDecoded,
                  ENV_TYPE env);
void PrintAsnAny (FILE* f, AsnAny* v, unsigned short indent);


/* AnyDefinedBy is the same as AsnAny */
typedef AsnAny AsnAnyDefinedBy;
#define FreeAsnAnyDefinedBy FreeAsnAny
#define BEncAsnAnyDefinedBy BEncAsnAny
#define BDecAsnAnyDefinedBy BDecAsnAny
#define PrintAsnAnyDefinedBy PrintAsnAny
```

4.13 Buffer Management

Encoding and decoding performance is heavily affected by cost of writing to and reading from buffers, thus, efficient buffer management is necessary. Flexibility is also important to allow integration of the generated encoders and decoders into existing environments. To provide both of these features, the calls to the buffer routines are actually macros that can be configured as you want (see `snacc/c_include/asn_config.h`). Virtually all buffer calls will be made from the encode/decode library routines. So macros used in the generated code will make buffer calls.

If your environment uses a single, simple buffer type, the buffer routine macros can be defined as the macros for your simple buffer type. This results in the buffer type being bound at compile time, with no function call overhead from the encode or decode routines. This also mean that the runtime library only works for that buffer type.

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

The backwards encoding technique requires special buffer primitives that write from the end of the buffer towards the front. This requirement will make it impossible to define buffer primitives that write directly to stream oriented objects such as TCP connections. In cases such as this, you must encode the entire PDU before sending it. (Or else extend the back-end of the compiler to produce "forwards" encoders as well).

Nine buffer primitives are required by the runtime library's encode and decode routines:

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

These buffer operations are described in the next subsections. The `ExpBuf`, `SBuf` and `MinBuf` buffer formats that come with the snacc distribution and how to configure the buffer operations are discussed following that.

4.13.1 Buffer Reading Routine Semantics

The buffer reading routines are called by the decoder routines. The following is the list of necessary buffer reading routines and their semantics. Be sure to setup the buffer in reading mode before calling any of these routines. The means of putting a buffer in reading mode depends on the buffer type.

```
unsigned char BufGetByte(BUF_TYPE b);
```

Returns the next byte from the buffer and advances current pointer such that a subsequent buffer read returns the following byte(s). This will set the read error flag if an attempt to read past the end of the data is made.

```
unsigned char BufPeekByte(BUF_TYPE b);
```

Returns the next byte from the buffer without advancing the current pointer.

```
char* BufGetSeg(BUF_TYPE b, unsigned long int* lenPtr);
```

Returns a pointer to the next bytes from the buffer and advances the current pointer. `*lenPtr` should contain the number of bytes to read. If the buffer has a least `*lenPtr` contiguous bytes remaining to be read before calling `BufGetSeg`, a pointer to them will be returned and `*lenPtr` will be unchanged. If there are less than `*lenPtr` contiguous bytes remaining in the buffer before the call to `BufGetSeg`, a pointer to them is returned and `*lenPtr` is set to the actual number of bytes that are referenced by the returned pointer. The current pointer will be advanced by the value returned in `*lenPtr` (this may advance to the next buffer segment if any). Note that the read error flag is not set if `*lenPtr` is greater than the remaining number of unread bytes.

```
unsigned long int BufCopy( char* dst, BUF_TYPE b, unsigned long int len)
```

Copies the next `len` bytes from the buffer into the `dst char*` and advances the current pointer appropriately. Returns the number of bytes actually copied. The number of bytes copied will be less than requested only if the end of data is reached, in which case the read error flag is set.

```
void BufSkip(BUF_TYPE b, unsigned long int len);
```

Advances the buffer's current pointer by len bytes. This will set the read error flag if less than `len` unread bytes remain in the buffer before the call to `BufSkip`.

```
int BufReadError(BUF_TYPE b);
```

Returns non-zero if a read error occurred for the given buffer. Read errors occur if one of the buffer reading routines attempted to read past the end of the buffer's data.

4.13.2 Buffer Writing Routine Semantics

Encoding routines call the buffer writing routines. Here is a list of the buffer writing routine and their semantics. Before calling the writing routines, you should make sure the buffer is setup for writing in reverse mode. The means of doing this depends on the buffer type.

```
void BufPutByteRvs(BUF_TYPE b, unsigned char byte);
```

Writes the given byte to the beginning of the data in the given buffer. The newly written byte becomes part of the buffer's data such that subsequent writes place bytes before the newly written byte. If a buffer write error occurs, subsequent writes do nothing.

```
void BufPutSegRvs(BUF_TYPE b, char* data, unsigned long int len);
```

Prepends the given bytes, data, of length len to the beginning of the data in the given buffer b. The data bytes are written such that the first byte in data becomes the first byte of the buffer's data, followed by the rest. (This means the bytes in data are not reversed, they are simply prepended as a unit to the buffer's original data). If a buffer write error occurs, subsequent writes do nothing.

```
int BufWriteError(BUF_TYPE b);
```

Returns non-zero if a write error occurred for the given buffer. Write errors occur if the buffer runs out of space for data or cannot allocate another data block (depends on the buffer type).

4.13.3 Buffer Configuration

The runtime library's encode and decode routines as well as the generated code access the buffers via the nine buffer macros described in the last two sections. These macros can be defined to call simple macros for speed or to call functions. Note that the buffer configuration is bound at the time the library and generated code are compiled.

The following is from snacc/include/asn_config.h and shows how to configure the buffer routines. This setup will make all calls to BufGetByte in the library and generated code call your ExpBufGetByte routine; the other buffer routines are mapped to their ExpBuf equivalents in a similar way.

```
#include "exp_buf.h"
#define BUF_TYPE ExpBuf**
#define BufGetByte(b)               ExpBufGetByte(b)
#define BufGetSeg(b, lenPtr)        ExpBufGetSeg(b, lenPtr)
#define BufCopy( dst, b, lenPtr)    ExpBufCopy( dst, b, lenPtr)
#define BufSkip( b, len)            ExpBufSkip(b, len)
#define BufPeekByte( b)             ExpBufPeekByte( b)
#define BufPutByteRv( b,  byte)     ExpBufPutByteRv( b, byte)
#define BufPutSegRv( b, data, len) ExpBufPutSegRv( b, data, len)
#define BufReadError(b)             ExpBufReadError(b)
#define BufWriteError(b)            ExpBufWriteError(b)
```

If you want to use your own buffer type, simply edit the asn_config.h file such that it includes your buffer's header file, sets the BUF_TYPE type, and defines the nine buffer routines (BufGetByte etc.) to call your buffer routines. Your buffer routines should have the semantics and prototypes described in the last two sections (Sections 4.13.1 and 4.13.2).

4.13.4 ExpBuf Buffers

The `ExpBuf` buffers are a doubly linked series of buffers that can be expanded when encoding by adding new buffers as necessary. Each `ExpBuf` consists of two blocks of memory, one for the control and linking information and the other for the data; when refering to an `ExpBuf` both parts are included. `ExpBuf` is short for "Expanding Buffer". Look in `snacc/c_lib/exp_buf.c` for an ASCII drawing of the `ExpBuf` buffers. Take a look a the `snacc/c_examples/simple/expbuf_ex.c` file for a quick introduction on how to use `ExpBufs`.

`ExpBufs` are fairly general and useful when a reasonable upper bound can not be put on the size of the encoded values that will be encountered by the protocol. The flexibility of these buffer routines will hurt the performance as many of the `ExpBuf` calls are not macros and new buffers may need to be allocated during encoding.

For encoding you need to write into the `ExpBufs`. Start with a single ExpBuf (or the last one in a list of ExpBufs from a previous encoding). Make sure this ExpBuf has been reset is "Write Reverse" mode (use `ExpBufResetInWriteRvsMode`). This clears the write error flag (and sets the read error flag in case you try a read) and resets the data start and data end pointers such that the buffer is empty and ready for writing from the end towards the front.

During encoding, if an `ExpBuf`'s data part fills up, a new `ExpBuf` before (since writing in reverse) the current buffer is needed. If the `prev` pointer in the current buffer is non-NULL, the previous buffer is reset for writing and becomes the current buffer. If the `prev` pointer in the current buffer NULL, a new buffer is allocated, its pointer is placed in `prev` and it becomes the current buffer. The notion of current buffer is handled by the parameter to the encoding and decoding routines. The buffer parameter is a an `ExpBuf**` and it always holds the current `ExpBuf*` (current buffer).

When encoding is finished and the encoded value has been transmitted, you have two options. You can free the entire buffer list or you can keep them around and re-use them for the next encoding. Freeing the buffers after each encoding may be quite slow. If you re-use the buffers, the buffer list will grow to the size of the largest encoding and stay there. You can easily implement other management schemes. By default the `ExpBufs` (both parts) are allocated and freed with `malloc` and `free`; you may want to change this to fit you environment better. If buffer allocation fails during a write, the writeError flag will be set and subsequent writes will do nothing.

For decoding you will want to put the encoded data into the `ExpBuf` format. For example, if your encoded value is contiguous in a single block of memory, you could use `ExpBufInstallDataInBuf` to attach your data to a single ExpBuf. Once your data is in the ExpBuf format, you should call `ExpBufResetInReadMode` on the first buffer in the list (if more than one). Then you can pass it to the desired decode routine.

If a decode routine attempts to read past the end of a buffer (usually due to an erroneous encoding), the readError flag will be set for the current `ExpBuf` in the list. This error will typically cause the decoding routine that called the buffer read routine to call `longjmp`.

The `BUF_TYPE` is defined as `ExpBuf**` so that the buffer parameter `b` can be set to the next active `ExpBuf` by the buffer routines. This saves having a head of the list type structure that keeps track of the the first, last and current buffers (the indirectness of this approach would hurt performance).

There are many routines for administrating the `ExpBufs` if you want to treat them like an abstract data type. Sometimes it may be easier to skip the utility routines and modify the fields directly.

The following routines are the required nine buffer routines. Compile the library and the generated code with the `USE_EXP_BUF` symbol defined to map buffer routines that the generated and library code calls to the `ExpBuf` routines (see `snacc/c_include/asn_config.h`). These `ExpBuf` routines adhere to the buffer routine prototypes and semantics defined in Sections 4.13.1 and 4.13.2.

```
void          ExpBufSkip(ExpBuf**, unsigned long len);
int           ExpBufCopy(char* dst, ExpBuf** b, unsigned long len);
unsigned char ExpBufPeekByte(ExpBuf** b);
char*         ExpBufGetSeg(ExpBuf** b, unsigned long* len);
void          ExpBufPutSegRvs(ExpBuf** b, char* data, unsigned long len);
unsigned char ExpBufGetByte(ExpBuf** b);
void          ExpBufPutByteRvs(ExpBuf** b, unsigned char byte);
#define       ExpBufReadError(b) ((*b)->readError)
#define       ExpBufWriteError(b) ((*b)->writeError)
```

The following `ExpBuf` routines are also provided. Their descriptions can be found in the code.

```
void ExpBufInit(unsigned long dataBlkSize);
void ExpBufInstallDataInBuf(ExpBuf* b, char* data, unsigned long int len);

void ExpBufResetInReadMode(ExpBuf* b);
void ExpBufResetInWriteRvsMode(ExpBuf* b);

ExpBuf* ExpBufAllocBufAndData();
void ExpBufFreeBufAndData(ExpBuf* b);
void ExpBufFreeBufAndDataList(ExpBuf* b);

ExpBuf* ExpBufNext(ExpBuf* b);
ExpBuf* ExpBufPrev(ExpBuf* b);
ExpBuf* ExpBufListLastBuf(ExpBuf* b);
ExpBuf* ExpBufListFirstBuf(ExpBuf* b);

int ExpBufAtEod(ExpBuf* b);
```

```
int ExpBufFull(ExpBuf* b);
int ExpBufHasNoData(ExpBuf* b);

char* ExpBufDataPtr(ExpBuf* b);
unsigned long ExpBufDataSize(ExpBuf* b);
unsigned long ExpBufDataBlkSize(ExpBuf* b);
```

4.13.5 SBuf Buffers

The SBufs are simple buffers of a fixed size, much like an ExpBuf that cannot expand. If you attempt to write past the end of the buffer, the writeError flag will be set and the encoding will fail. If you attempt to read past the end of a buffer the readError flag will be set and the decoding will fail.

The SBufs are useful if you can put a reasonable upper bound the size of the encodings you will dealing with. The buffer operations are much simpler because the data is contiguous. In fact, all of the SBuf buffer operations are implemented by macros.

Look in snacc/c_examples/simple/sbuf_ex.c for a quick introduction to using SBufs in your code. The following operations are defined for the SBuf buffers.

```
/* The nine required buffer operations */
#define SBufSkip(b, skipLen) ...
#define SBufCopy(dst, b, copyLen) ...
#define SBufPeekByte(b) ...
#define SBufGetSeg( b, lenPtr) ...
#define SBufPutSegRvs(b, seg, segLen) ...
#define SBufGetByte(b) ...
#define SBufPutByteRvs(b, byte) ...
#define SBufReadError(b) ...
#define SBufWriteError(b) ...

/* other useful buffer operations */
#define SBufInit(b, data, dataLen) ...
#define SBufResetInReadMode(b) ...
#define SBufResetInWriteRvsMode(b) ...
#define SBufInstallData(b, data, dataLen) ...
#define SBufDataLen(b) ...
#define SBufDataPtr(b) ...
#define SBufBlkLen(b) ...
#define SBufBlkPtr(b) ...
#define SBufEod(b) ...
```

Snacc is configured to use SBufs by default. The symbols that will affect the buffer configuration during compilation of the libraries and generated code are USE_EXP_BUF and

USE_MIN_BUF.

4.13.6 MinBuf Buffers

The MinBufs provide maximum performance but should only be used under restricted conditions (to avoid segmentation faults etc.). No checks are made to determine whether a decoder is reading past the end of the buffer or if an encoder is writing "past" the beginning of the data block (remember, snacc encoders write backwards).

A MinBuf is just a char**; the referenced char* points to the next byte to be read or the last byte that was written. The read routine advance the char* and the write reverse routines move the char* backwards.

When you start encoding, the MinBuf char** should be a pointer to a pointer to the byte AFTER the last valid byte in your buffer. For example the following C fragment would work:

```
PersonnelRecord pr;
char blk[128];
char* minBuf;

minBuf = blk + 128; /* start writing a end of block */
BEncPersonnelRecord(&minBuf, pr);
```

The MinBufs should only be used during encoding if the size of the MinBuf's buffer is guaranteed to be large enough to hold the encoded value. Otherwise, the encoder will blindly continue writing into whatever lies after the MinBuf's buffer.

When you start decoding, the MinBuf value should be a pointer to a pointer the first byte of the BER value to be decoded. Look in snacc/c_examples/simple/minbuf_ex.c for a real example.

The MinBufs should only be used for decoding when the value being decoded is certain to contain no encoding errors. Otherwise, for encodings that are incomplete or contain length errors the decoder may attempt to read the memory that follows the MinBufs. If you are lucky, the decoder will return an error with the longjmp mechanism. If your system has memory protection and you are unlucky this may abort your program. If you are really unlucky, the data following the MinBuf may fool the decoder into thinking that it is valid and you receive a wrong PDU with no error indication. This risky technique has been used successfully in some systems where the encodings are not guaranteed to be correct.

To configure the generated code to use the MinBufs, compile it with the USE_MIN_BUF symbol defined.

4.13.7 Hybrid Buffer Solutions

The decoding routines only call the buffer reading routines and the encoding routines only call the buffer writing routines. You may wish to choose different a buffer format for the encoding and decoding to gain performance. For instance, if you can be sure that the size of outgoing encodings is less than a certain upper bound, but don't want to risk segmentation faults when decoding incoming values, you could use `MinBufs` for the the buffer writing (encoding) operations and `SBufs` or `ExpBufs` for the buffer reading (decoding) operations.

In this case you will need to massage the generated code to achieve the desired results.

4.14 Dynamic Memory Management

Like buffer management, efficient memory management is very important for efficient decoders. As a decoder decodes a value, it allocates memory to hold the internal representation of the value.

The runtime library's and the generated decode routines allocate memory using the `Asn1Alloc` routine. The runtime library's and the generated free routines free memory using the `Asn1Free` routine. The decoding routines also use `CheckAsn1Alloc` to make sure that each allocation succeeded. These memory routine are defined in the `asn_config.h` and have the prototypes:

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

The decoders assume that `Asn1Alloc` returns a *zeroed* block of memory. This saves explicit initialization of OPTIONAL elements with NULL in the generated decoders. It wouldn't be too hard to modify the compiler to produce decoders that initialized OPTIONAL elements explicitly.

The generated free routines hierarchically free all for a value's memory using depth first algorithm. If you use the Nibble Memory scheme, you will not need the generated free routines.

By default, snacc uses a "Nibble Memory" scheme to provide efficient memory management. Nibble Memory works by allocating a large block of memory for allocating from. When the decoded value has been processed, you can free the entire value by calling a routine that simply resets a few pointers. There is no need to traverse the entire value freeing a piece at a time. The following is from `nibble_alloc.h`.

```
void InitNibbleMem(unsigned long int initialSize,
                   unsigned long int incrementSize);
void* NibbleAlloc(unsigned long int size);
```

```
void ResetNibbleMem();
void ShutdownNibbleMem();
```

You must explicitly initialize the Nibble Memory with the `InitNibbleMem` routine before using a decoder. You must specify the initial size of the nibble block and the size that it should grow by. If you attempt to allocate a block that is larger that the initial nibble block or its grow size, a new block of the correct size will be allocated. Note that the "growth" occurs by linking separate blocks, not by the potentially slow alternative, `realloc`.

When you have processed the decoded value you can free it by calling `ResetNibbleMem`. This resets a couple pointers and frees any extra blocks that were allocated to handle values larger than the initial block size. The original memory block is zeroed using `memset` so that all allocations will return zeroed values. This is necessary to support the implicit initialization of OPTIONAL elements to NULL. The zeroing is done in this routine instead of `NibbleAlloc` under the assumption that zeroing one large block is more efficient than zeroing pieces of it as they are allocated.

When you no longer need the Nibble Memory, you can release it by using `ShutDownNibbleMem`. This frees all of the memory associated with Nibble Memory, both the control data and the block(s) used for allocation.

There are some problems with this memory management scheme. Currently the Nibble Memory control information is kept track of via a global variable that holds a pointer to the control information. This can present a problem is separate Nibble Memory contexts are needed, for example, one to hold one value that will be kept after decoding and another to hold a decoded value that will soon be discarded.

The problem of separate contexts could be solved by adding another layer that would use identifiers for different memory contexts. This would require you to set the context using its identifier before calling a decoding routine and to pass the context identifier to the `ResetNibbleMem` routine.

Another problem has to do with building the values to be encoded. There is no restriction on what allocator you use to build internal values. However, it is convenient to use the `AsnListNew` routine to allocate and initialize a list type. Unfortunately, `AsnListNew` is used by the decoding routines so it uses the `Asn1Alloc` routine to allocate the new list. You should be aware of this if `Asn1Alloc` is not what you are using to allocate the rest of the value. This could be fixed with a different interface to the `AsnListNew` routine.

It is possible to change the memory management system without to much difficulty. For example if you are not too worried about performance and want to use `malloc` and `free`, you could change the `asn_config.h` file as follows:

```
#include "malloc.h"
#define Asn1Alloc(size)  calloc(1, size)
#define Asn1Free(ptr)    free(ptr)
#define CheckAsn1Alloc(ptr, env)\
   if ((ptr) == NULL)\
```

```
longjmp(env, -27);
```

If you use `malloc` based allocators such as `calloc`, you must use the generated free routines to free your values. Note that this example used `calloc` instead of `malloc` because `calloc` *zeroes* each allocated block of memory, as required by the decoders.

4.15 Error Management

The decoding routines use `longjmp` to handle any errors they encounter in the value being decoded. `longjmp` works by rolling back the stack to where the `setjmp` call was made. Every decode routine takes a `jmp_buf env` parameter (initialized by the `setjmp` call) that tells the `longjmp` routine how to restore the processor to the correct state. `longjmp` makes the error management much simpler since the decoding routines do not have to pass back error codes or check ones from other decoding routines.

Before a PDU can be decoded, the `jmp_buf env` parameter to the decoding routine must be initialized using the `setjmp` routine. This should be done immediately and only once before calling the decoding routine. This parameter will be passed down to any other decoding routines called within a decoding routine. The following code fragment from `snacc/c_examples/simple/exbuf_ex.c` shows how to use `setjmp` before decoding.

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

The code that will signal an error typically looks like:

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

Most `longjmp` calls are preceded by a call to `Asn1Error` which takes a single `char*` string as a parameter. The library routines and the generated code try to use meaningful messages as the parameter. `Asn1Error` is defined in `asn_config.h` and currently just prints the given string to `stderr`. You may wish to make it do nothing, which may shrink the size of your binary because all of the error strings will be gone. `Asn1Warning` is similar but is not used by the library or generated code anymore.

The encoding routines do no error checking except for buffer overflows. Hence, they do not use the `longjmp` mechanism and instead require you to check the status of the buffer after encoding (use `BufWriteError()`). If you are not building your values properly, for example having random pointers for uninitialized OPTIONAL elements, the encode routines will fail, possibly catastrophically.

Chapter 5

C++ Code Generation

5.1 Introduction

The C++ backend of snacc was designed after the C backend had been written. The basic model that the generated C++ uses is similar to that of the generated C, but benefits from the object oriented features of C++. This was my first real foray into C++ which may be evident from some of the design.

As with C, two files are generated for each ASN.1 module, a `.C` and a `.h` file.

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

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

For error management C++'s `try` and `catch` are obvious replacements for the `setjmp` and `longjmp` used by the C decoders. Unfortuantely this is a newer C++ feature and is not yet supported by gcc.

C++ templates are very attractive for type safe lists (for SET OF and SEQUENCE OF) without duplicating code. Template support was shaky in gcc at the time the generated code was being tested so they were rejected. Instead, each list generates its own new class with all of the standard list routines.

As with the C code generation chapter, we will use the EX1 module to help illustrate some of the code generation. The following is the same EX1 module used in the C section.

```
EX1 DEFINITIONS ::=
BEGIN

anOidVal      OBJECT IDENTIFIER ::= { joint-iso-ccitt 40 foobar(29) }
theSameOidVal OBJECT IDENTIFIER ::= { 2 40 29 }
```

```
anIntVal       INTEGER ::= 1
aBoolVal       BOOLEAN ::= TRUE

T1 ::= SEQUENCE
{
   INTEGER OPTIONAL,
   OCTET STRING OPTIONAL,
   ENUMERATED { a(0), b(1), c(2) },
   SEQUENCE OF INTEGER,
   SEQUENCE { id OBJECT IDENTIFIER, value OCTET STRING },
   CHOICE { INTEGER, OBJECT IDENTIFIER }
}

END
```

The C++ backend to snacc is in the `snacc/src/back_ends/c++_gen` directory if you want to alter it. Warning! the backend code is pretty ugly.

5.2 ASN.1 to C++ Naming Conventions

The C++ name for a type or value is the same as its ASN.1 name with any hyphens converted to underscores.

When an ASN.1 type or value name (after converting any hyphens to underscores) conflicts with a C++ keyword or the name of a type in another ASN.1 module (name clashes within the same ASN.1 scope are considered errors and are detected earlier), the resulting C++ class name will be the conflicting name with digits appended to it.

Empty field names in SETs, SEQUENCEs, and CHOICEs will be filled. The field name is derived from the type name for that field. The library types such as INTEGER etc. have default field names defined by the compiler (see `snacc/src/back_ends/c_gen/rules.c` and `snacc/src/back_ends/cpp_rules.c`). The first letter of the field name is in lower case. Empty field names should be fixed properly by adding them to the ASN.1 source.

New type definitions will be generated for SETs, SEQUENCEs, CHOICEs, ENUMERATED, INTEGERs with named numbers and BIT STRING with named bits whose definitions are embedded in other SET, SEQUENCE, SET OF, SEQUENCE OF, or CHOICE definitions. The name of the new type is derived from the name of the type in which it was embedded and will be made unique by appending digits if necessary.

5.3 ASN.1 to C++ Class Translation

This section describes how C++ classes are used to represent each ASN.1 type. First the general characteristics of each ASN.1 type's C++ class will be discussed followed by how the aggregate types (SETs, SEQUENCEs, CHOICEs, SET OFs, and SEQUENCE OFs) are represented. The representations of non-aggregate types (INTEGER, BOOLEAN, OCTET STRING, BIT STRING, OBJECT IDENTIFIER) and ANY and ANY DEFINED BY types are presented in the next chapter since they form part of the C++ ASN.1 runtime library.

Every ASN.1 type is represented by a C++ class with the following characteristics:

1 inherits from the `AsnType` base class

2 has a parameterless constructor

3 has a clone method, `Clone`

4 has a PDU encode and decode method, `BEnc` and `BDec`

5 has a content encode and decode method, `BEncContent` and `BDecContent`

6 has top level interfaces to the PDU encode and decode methods (handles the `setjmp` etc.) for the user, `BEncPdu` and `BDecPdu`

7 has print method, `Print`

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

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

  public:
    Foo();  // requires parameterless constructor

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

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

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

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

`BEnc` and `BDec` are PDU encode and decode methods. `BEnc` encodes the tag and length pairs for the object's type as well as the content (the object's value) to the given buffer, `b`, and returns the number of bytes written to the buffer for the encoding.

BDec decodes the expected tag and length pairs as well as the content of the object it is invoked upon from the given buffer, b, and increments bytesDecoded by the byte length of the tag(s), length(s) and value decoded. The env parameter will be used with longjmp if any decoding error occurs. Decoding errors can be reported via longjmp from any of the routines that BDec calls, such as BDecContent; BDec will call longjmp directly if the value does not have the correct tag(s).

BEncContent and BDecContent only deal with the content of the type their object represents. BEncContent encodes the object's value to the given buffer, b.

BDecContent decodes the object's value from the given buffer, b. The last tag and length pair on the content must be passed in via the the tag and elmtLen parameters. The tag, although always present, will only be used when decoding OCTET STRING and BIT STRING related types, to determine whether the encoding is constructed. The elmtLen is the length of the content and may be the indefinite length form. bytesDecoded is incremented by the actual number of bytes in the content; this is normally the same as elmtLen unless the indefinite length form was decoded. The env parameter will be used with longjmp if any decoding error occurs. The possible decoding errors depend on the type that is being decoded.

BEncPdu and BDecPdu are top level interfaces to the PDU encode and decode routines. They present the simplest interface; they return TRUE if the operation succeeded and FALSE if an error occurred. Note that the BDecPdu routine sets up the env parameter using setjmp for any longjmp calls that may occur. If you call BDec or BDecContent directly from your code, you must use setjmp to setup the env parameter. BEncPdu checks for any buffer writing errors and BDecPdu checks for any buffer reading errors.

The Print method prints the object's value in ASN.1 value notation. When printing SETs and SEQUENCEs, an global variable is used for the current indent.

The AsnType base class, parameterless constructor and Clone method are required by the ANY and ANY DEFINED BY type handling mechanism explained in Sections 6.4 and 6.13. In brief, the AsnType provides a base type that has virtual BEnc, BDec and Clone routines. The Clone routine is used to generate a new instance (not a copy) of the object that it is invoked on. This allows the ANY DEFINED BY type decoder to create an new object of the correct type from one stored in a hash table, when decoding (the Clone routine calls the parameterless constructor). The virtual BEnc and BDec are called from AsnAny BEnc and BDec methods.

5.3.1 SET and SEQUENCE

SET and SEQUENCE types generate classes that have their components as public data members. This makes accessing the components similar to referencing the fields of a C struct. For example the T1 type in module EX1 will produce the following C++ class:

```
class T1 : public AsnType
```

```
{
  public:
    AsnInt* integer;
    AsnOcts* octs;
    T1Enum t1Enum;
    T1SeqOf t1SeqOf;
    T1Seq* t1Seq;
    T1Choice* t1Choice;

    T1()
    {
        /* init optional/default elements to NULL */
        integer = NULL;
        octs = NULL;
    }

    AsnLen BEnc(BUF_TYPE b);
    void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env);

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

    int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded);
    int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded);

    void Print(ostream& os);
    AsnType* Clone() { return new T1; }
};
```

All OPTIONAL components in a SET or SEQUENCE are referenced by pointer. The constructor will automatically set OPTIONAL fields to NULL. The other methods are as described at the beginning of this section.

SETs and SEQUENCEs must contain all non-OPTIONAL components and SEQUENCEs must be ordered, otherwise an error is reported. Tagging errors are also reported. All detected errors abort the decoding process via `longjmp`.

5.3.2 CHOICE

Each CHOICE type generates a class that has an anonymous union to hold the components of the CHOICE and a `choiceId` field to indicate which component is present.

Anonymous (un-named) unions allow you to reference the choice components with just the field name of the component; this makes referencing the contents of a CHOICE the same a referencing the contents of a SET or SEQUENCE.

The `choiceId` field contains a value in the `ChoiceIdEnum` that indicates the CHOICE field that is present. The names in the enumeration are derived from the field names of the

CHOICE components.

When building a local value to be encoded, you must be sure to set the `choiceId` such that it corresponds the value in the union. The decoder will set the `choiceId` when decoding incoming values.

Tagging errors are reported and abort the decoding process via `longjmp`.

The following C++ class is produced for the CHOICE in the EX1 module.

```
class T1Choice : public AsnType
{
  public:
    enum ChoiceIdEnum
    {
        integerCid = 0,
        oidCid = 1
    };
    enum ChoiceIdEnum choiceId;
    union ChoiceUnion
    {
        AsnInt* integer;
        AsnOid* oid;
    };

    T1Choice() {}

    AsnLen BEnc(BUF_TYPE b);
    void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env);

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

    int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded);
    int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded);

    void Print(ostream& os);
    AsnType* Clone() { return new T1Choice; }
};
```

5.3.3 SET OF and SEQUENCE OF

Each SET OF and SEQUENCE OF type produces its own list class, unlike the C backend which uses a single generic list type for all lists. This makes the C++ list routines type safe which allows the C++ compiler to detect more programmer errors.

C++ templates should be used to reduce the code duplication when they become widespread and reliably implemented. The duplicated list handling methods may bloat the size of the generated code.

Any tagging errors are reported and abort the decoding process via `longjmp`.

From the EX1 ASN.1 module the following list is produced:

```
class T1SeqOf : public AsnType
{
  protected:
    unsigned long int count;
    struct AsnListElmt
    {
        struct AsnListElmt* next;
        struct AsnListElmt* prev;
        AsnInt* elmt;
    } *first, *curr, *last;

  public:
    T1SeqOf();
    void SetCurrElmt(unsigned long int index);
    unsigned long int GetCurrElmtIndex();
    void SetCurrToFirst();
    void SetCurrToLast();

    // reading member fcns
    int Count();
    AsnInt* First();
    AsnInt* Last();
    AsnInt* Curr();
    AsnInt* Next();
    AsnInt* Prev();

    // routines that move the curr elmt
    AsnInt* GoNext();
    AsnInt* GoPrev();

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

    // write & alloc & copy - returns list after copying elmt
    T1SeqOf& AppendCopy(AsnInt& elmt);  // add elmt to end of list
    T1SeqOf& PrependCopy(AsnInt& elmt); // add elmt to beginning of list
    T1SeqOf& InsertBeforeAndCopy(AsnInt& elmt); // insert elmt before current elmt
    T1SeqOf& InsertAfterAndCopy(AsnInt& elmt); // insert elmt after current elmt

    // encode and decode routines
    AsnLen BEnc(BUF_TYPE b);
    void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env);
```

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

    int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded);
    int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded);

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

Each list is doubly linked to allow simple reverse traversal for backwards encoding. The `prev` pointer will be NULL for the first element of the list and the `next` pointer will be NULL for the last element of the list.

Each list maintains a pointer the current element of the list. Several routines are provide to manipulate the current item. Since there is only one current pointer, you may have to save and restore the current pointer with the `GetCurrElmtIndex` and `SetCurrElmt` methods if you call routines that deal with the list while iterating through it.

In addition to the standard encode, decode and print methods some list utility routines are included in each list class. They are fairly simple and are described briefly here.

```
void SetCurrElmt(unsigned long int index);
```

This sets the current pointer the element with the given index. Indexes start at zero, that is, the first element in the list has an index of zero. If the given index is greater than or equal the number of elements in the list, the current pointer is set to the last element of the list.

```
unsigned long int GetCurrElmtIndex();
```

This returns the index of the current element. If the current pointer is NULL (or does not reference an element of the list, which is an error condition), the index returned will be greater than or equal the the number of elements in the list (indexes start at zero so this is an invalid index).

```
void SetCurrToFirst();
```

This sets the current pointer to the first element of the list. If the list is empty, it is set to NULL.

```
void SetCurrToLast();
```

This sets the current pointer to the last element of the list. If the list is empty, it is set to NULL.

```
int Count();
```

This returns number of elements in the list.

```
AsnInt* First();
AsnInt* Last();
AsnInt* Curr();
AsnInt* Next();
AsnInt* Prev();
```

The above routines return a pointer to the list element that the routine name indicates. They return NULL if the requested element is not present. For example `First` will return a pointer to the first element in the list or NULL if the list is empty. These routines do not affect the state of the list; the current pointer and the count remain the same.

```
AsnInt* GoNext();
AsnInt* GoPrev();
```

These routines change the current pointer to the next/previous element and return a pointer to that element. If the current element is NULL or points to the last element, `GetNext` returns NULL. Similarly, if the current element is NULL or points to the first element, `GetPrev` returns NULL.

```
AsnInt* Append();
```

This allocates a new list element, appends it to the end of list and returns a pointer to the new list element. Notice that you must set the value of the returned list element.

```
AsnInt* Prepend();
```

This allocates a new list element, prepends it to the beginning of list and returns a pointer to new the list element. You must set the value of the returned list element.

```
AsnInt* InsertBefore();
```

This allocates a new list element, inserts it before the current list element and returns a pointer to new the list element. You must set the value of the returned list element. If the current pointer is NULL, the new element is placed at the beginning of the list.

```
AsnInt* InsertAfter();
```

This allocates a new list element, inserts it after the current list element and returns a pointer to new the list element. You must set the value of the returned list element. If the current pointer is NULL, the new element is placed at the end of the list.

```
T1SeqOf& AppendCopy(AsnInt& elmt);
T1SeqOf& PrependCopy(AsnInt& elmt);
T1SeqOf& InsertBeforeAndCopy(AsnInt& elmt);
T1SeqOf& InsertAfterAndCopy(AsnInt& elmt);
```

These are similar to the `Append`, `Prepend`, `InsertBefore` and `InsertAfter` routines except that a copy of the given element's value is placed in the list and the list itself is returned.

5.3.4 ENUMERATED, Named Numbers and Named Bits

The C++ type generator encapsulates each ENUMERATED type, INTEGER with named numbers and BIT STRING with named bits in a new class that inherits from the proper base class and defines the named elements. This provides a separate scope for these identifiers so their symbol will be exactly the same as their ASN.1 counterpart. Currently these identifiers are not checked for conflicts with C++ keywords, so you may have to modify some of them in the ASN.1 modules.

Inheritance is used for attaching ENUMERATED, named number and named bit information. ENUMERATED types inherit from the `AsnEnum` class, INTEGERs with named number types inherit from the `AsnInt` class and BIT STRINGs with named bits inherit from the `AsnBits` class.

If the tagging on the type is different from the type it inherits from, the PDU encode and decode methods are re-defined with the correct tags to override the PDU encode and decode methods of the base class.

As with the other types, any tagging errors are reported and abort the decoding process via `longjmp`. No range checking is done on the decoded values although it would be easy to provide an new `BDecContent` method in the new class that calls the base class's and then checks the range of the result.

```
/* ENUMERATED { a(0), b(1), c(2) }  */
class T1Enum : public AsnEnum
{
  public:
    T1Enum(): AsnEnum() {}
    T1Enum(int i): AsnEnum(i) {}
    enum
    {
```

```
        a = 0,
        b = 1,
        c = 2
    };
};
```

5.4 ASN.1 to C++ Value Translation

C++ `const` values are used to hold ASN.1 defined values. C++ values will be produced for INTEGER, BOOLEAN and OBJECT IDENTIFIER ASN.1 values. An `extern` declaration for each `const` value is written at the end of the header file of the value's module. The `const` values are defined at the beginning of the `.C` file of the value's module. The `extern` declarations are at the end of the header file so that any required class definitions are available.

The following is from the end of the header file generated for the EX1 module:

```
extern const AsnOid anOidVal;
extern const AsnOid theSameOidVal;
extern const AsnInt anIntVal;
extern const AsnBool aBoolVal;
extern const AsnInt foobar;
```

The following is from the beginning of the `.C` file generated for the EX1 module:

```
const AsnOid anOidVal(2, 40, 29);
const AsnOid theSameOidVal(2, 40, 29);
const AsnInt anIntVal(1);
const AsnBool aBoolVal(AsnBool::true);
const AsnInt foobar(29);
```

The C++ constructor mechanism is used to generate these values. This mechanism is superior to C static initialization because it allows C++ code to be run to initialize the values.

5.5 Compiler Directives

Compiler directives are ignored by the C++ backend of snacc. If you want to implement them, look at the `FillCppTypeDefInfo` routine in file `snacc/src/back_ends/c++_gen/cpp_types.c`. Then look at the way it is done for the C backend (file `snacc/src/back_ends/c_gen/type_info.c`)

5.6 Compiling the Generated C++ Code

When compiling the generated C++ code you will need:

1 `snacc/c++_include` directory in your include path so the C++ sources can include the library header files. (`-I snacc/c++_include`)

2 to link with the C++ ASN.1 runtime library, `snacc/c++_lib/libasn1c++.a`.

3 to link with the math library, since the ASN.1 REAL type's encode and decode routine use some math routines. (`-lm`)

See the example in `snacc/c++_examples/simple` for a complete example. The makefile and main routines are probably the most important. There are several other examples in the `snacc/c++_examples` directory.

Chapter 6

C++ ASN.1 Library

6.1 Overview

The following sections describe the C++ representation of the non-aggregate ASN.1 types, ANY and ANY DEFINED BY types and the buffer and memory management. These classes and routines make up the C++ ASN.1 runtime library. Every aggregate ASN.1 type will be composed of these library types. The source files for this library are in `snacc/c++_include` and `snacc/c++_lib`.

As mentioned in the last chapter, each ASN.1 type is represented by a C++ class which inherits from the `AsnType` base class. In addition to the standard encode, decode, print and clone methods described in the last chapter, each ASN.1 type class in the library may also have special constructors and other routines that simplify their use.

Unlike the classes generated for some of the aggregate types such as SETs and SEQUENCEs, the library types data members are typically protected and accessed via methods.

All of the library classes' `BDec` routines will report tagging errors via `longjmp` as described in Section 5.3.

The top level PDU encode and decode methods are the same for all library types so they are defined as macros in `snacc/c++_inlcude/asn_config.h`. For clarity's sake, the macro that is used to define these methods in the library type class definitions will be replaced with the actual prototypes.

Run the `test_lib` program in `snacc/c++_examples/test_lib` to make sure the library routines are working properly for your architecture. The testing is not exhaustive but should point out obvious problems.

6.2 Tags

The C++ tags are identical those used in snacc's C ASN.1 environment. While it would have been nice define a tag class, the performance cost would likely have been noticeable. Also, the snacc users usually do not have to muck around with tags directly, so the lack of a class interface will probably not be missed. The C ASN.1 tags are described in Section 4.2.

Initially I defined a C++ class for tags, but close examination of the produced assembly code led me to reject it. The C++ class for tags used the C tag representation internally and had constructor, encode and decode methods. The constructor could not be used as `switch` statement case labels like `MAKE_TAG_ID` because it did not reduce to an integer constant; this caused problems in the generated decoders.

As with the C representation of tags, 4 byte long integers limit the maximum representable tag code to 2^{21}. Again, this should not be a problem.

6.3 Lengths

The C++ representation of lengths is the same at the C representation described in Section 4.3. The length type was not given its own C++ class for reasons similar to those of tags.

6.4 The AsnType Base Class

Every ASN.1 type's C++ class uses the `AsnType` as its base class. The `AsnType` base class is provides the following virtual functions:

- a destructor
- `BDec`
- `BEnc`
- `Print`
- `Clone`

The `AsnType` class is defined as follows:

```
class AsnType
{
  public:

    virtual ~AsnType() {}
```

```
#ifdef SUPPORT_ANY_TYPE
    virtual void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env);
    virtual AsnLen BEnc(BUF_TYPE b);
    virtual AsnType* Clone();
#else
    void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env) {}
    AsnLen BEnc(BUF_TYPE b) { return 0;}
#endif
    virtual void Print(ostream& os);
};
```

The `AsnType` class and its virtual functions were added to support the ANY DEFINED BY type handling mechanism. This mechanism is described in Section 6.13.

Even if you do not use the ANY or ANY DEFINED BY types, the `AsnType` base class may be useful for adding features that are common to all of the types, such as changing the `new` and `delete` functions to improve performance.

Virtual functions provide the simplest method of handling ANY DEFINED BY and ANY types. Unfortuanately, calls to virtual functions are slower than calls to normal functions due their indirect nature. If you do not need support for the ANY DEFINED BY or ANY types you can remove most of the virtual functions to improve performance by undefining the `SUPPORT_ANY_TYPE` symbol (see the `asn_type.h` file).

Note that a virtual destructor is included in the `AsnType` base class as well. This is done to make sure the `delete` routine always gets the correct size. See pp 215-217 of Stroustrup [8] for a discussion of this.

6.5 BOOLEAN

The BOOLEAN type is represented by the `AsnBool` class. The following is the class definition of `AsnBool` from the `snacc/c++_include/asn_bool.h` file.

```
class AsnBool: public AsnType
{
  protected:
    unsigned char value;

  public:
    enum { false = 0, true = 1};
    operator int() { return value; }
    AsnBool(const int val): value(val != false) {};
    AsnBool() {};

    AsnLen BEnc(BUF_TYPE b);
    void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env);
```

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

    int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded);
    int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded);

    void Print(ostream& os);
    AsnType* Clone() { return new AsnBool; }
};
```

The `AsnBool` class defines the enumerated values of `true` and `false` if you want to check to a boolean's value explicitly. Their enum values are defined inside the `AsnBool` class to avoid potential conflicts with other symbols. However, they must by referenced by `AsnBool::true` and `AsnBool::false`.

The `operator int()` is defined such that when an `AsnBool` value is cast to an integer, it returns the C style integer boolean value of the `AsnBool`s value. That is, zero for false and non-zero for true. There is also a constructor for `AsnBool` that builds an `AsnBool` value from the given C style integer boolean value. These two methods allow you to manipulate and access `AsnBool` values in a straight forward way as the following code illustrates.

```
Message::Send()
{
    AsnBool okToSend;
    int connectionOpen;
    int  pduOk;
    . . .
    okToSend = connectionOpen && pduOk; // assign AsnBool from integer bool
    if (okToSend) // cast AsnBool to integer boolean
      . . .
}
```

The `AsnBool` class contains the standard encode and decode methods that were described in chapter 5.

`BDecContent` will report an error via `longjmp` if the length of an encoded BOOLEAN value's content is not exactly 1 octet.

Note that the `Clone` method returns an `AsnType*` value instead of an `AsnBool*`. It might be more obvious to return an `AsnBool*` since due to single inheritance an `AsnBool` is also an `AsnType`. However, it must return an `AsnType*` for it to override the virtual function `Clone` defined in the `AsnType`.

The `Print` method will print either `TRUE` or `FALSE` depending on the `AsnBool` value. No newline or other formatting characters are printed. The global indent information does not affect the output from this method.

6.6 INTEGER

The INTEGER type is represented by the `AsnInt` class. The following is the class definition of `AsnInt` from the `snacc/c++_include/asn_int.h` file.

```
class AsnInt : public AsnType
{
  protected:
    long int value;

  public:
    operator long int() { return value; }
    AsnInt(long int val): value(val) {}
    AsnInt() {}

    AsnLen BEnc(BUF_TYPE b);
    void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env);

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

    int BEncPdu(BUF_TYPE b, AsnLen& bytesEncoded);
    int BDecPdu(BUF_TYPE b, AsnLen& bytesDecoded);

    void Print(ostream& os) { os << value; }
    AsnType* Clone() { return new AsnInt; }
};
```

The internal representation of an ASN.1 INTEGER value is a `long int`. This `BDecContent` routine will signal an error if the integer value being decoded will not fit into the `long int` representation.

Unlike the C ASN.1 library, the `unsigned long int` version of INTEGER is not provided. If you need it, it should be relatively trivial to combine the C unsigned version with the existing C++ class. The unsigned version of an integer is useful is your ASN.1 source uses subtyping similar to:

```
Counter ::= [APPLICATION 1] IMPLICIT INTEGER (0..4294967295)
```

6.7 NULL

The NULL type is provided by the `AsnNull` class. This class has no data members and includes only the standard methods.

```
class AsnNull : public AsnType
{
```

```
  public:
    AsnNull() {}
    AsnLen BEnc( BUF_TYPE b);
    void BDec( BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env);

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

    int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded);
    int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded);

    void Print(ostream& os) { os << "NULL"; }
    AsnType* Clone() { return new AsnNull; }
};
```

6.8 REAL

REAL types are represented by the `AsnReal` class. Internally, a `double` is used to hold the real value. The following is from `snacc/c++_include/asn_real.h`:

```
class AsnReal: public AsnType
{
  protected:
    double value;

  public:
    operator double() const  { return value; }
    AsnReal(): value(0.0) {}
    AsnReal(double val): value(val) { }

    void Print(ostream& os) { os << value; }
    AsnType* Clone() { return new AsnReal; }

    AsnLen BEnc( BUF_TYPE b);
    void BDec( BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env);

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

    int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded);
    int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded);
};

extern const AsnReal PLUS_INFINITY;
extern const AsnReal MINUS_INFINITY;
```

The double representation and support routines can depend on the compiler or system you are using so several different encoding routines are provided. Even so, you may need to modify the code.

There are three content encoding routines included and they can be selected by defining one of IEEE_REAL_FMT, IEEE_REAL_LIB or nothing. Defining IEEE_REAL_FMT uses the encode routine that assumes the double representation is the standard IEEE double [3]. Defining IEEE_REAL_LIB uses the encode routine that assumes the IEEE functions library (isinf, scalbn, signbit etc.) is available. If neither are defined, the default encode routine uses frexp. Look at the related comments in snacc/c++_include/asn_config.h.

AsnReal constants are used to hold PLUS_INFINITY and MINUS_INFINITY values. These values are initialized using the AsnReal constructor mechanism with the AsnPlusInfinity and AsnMinusInfinity routines. If you do not define IEEE_REAL_FMT or IEEE_REAL_LIB, you should rewrite the AsnPlusInfinity routine such that it is correct for your system.

There is only one content decoding routine and it builds the value through multiplication and the pow routine (requires the math library). The content decoding routine only supports the binary encoding a of REAL, not the decimal encoding.

6.9 BIT STRING

The BIT STRING type is represented by the AsnBits class. From snacc/c++_include/asn_bits.h:

```
class AsnBits: public AsnType
{
  protected:
    char* bits;
    unsigned long int bitLen;

  public:
    AsnBits() { bits = NULL; bitLen = 0; }
    AsnBits(const unsigned long int numBits) { Set(numBits); }
    AsnBits(const char* bitOcts, const unsigned long int numBits)
    {
        Set(bitOcts, numBits);
    }
    AsnBits(const AsnBits& b) { Set(b); }
    ~AsnBits() {delete bits;}

    AsnBits& operator=(const AsnBits& b) { ReSet(b); return *this;}

    // overwrite existing bits and bitLen values
    void Set(unsigned long int numBits);
    void Set(const char* bitOcts, unsigned long int numBits);
    void Set(const AsnBits& b);
```

```
    // free old bits value, the reset bits and bitLen values
    void ReSet(const unsigned long int numBits);
    void ReSet(const char* bitOcts, unsigned long int numBits);
    void ReSet(const AsnBits& b);

    operator==(AsnBits& ab)  { return(BitsEquiv(ab)); }
    operator!=(AsnBits& ab)  { return(!BitsEquiv(ab)); }

    void SetBit(unsigned long int);
    void ClrBit(unsigned long int);
    int  GetBit(unsigned long int);

    unsigned long int BitLen() {return bitLen;}

    AsnLen BEnc( BUF_TYPE b);
    void BDec( BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env);

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

    int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded);
    int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded);

    AsnType* Clone() { return new AsnBits; }
    void Print(ostream& os);

  private:
    int BitsEquiv(AsnBits& ab);
    void BDecConsBits (BUF_TYPE b, AsnLen elmtLen,
                                   AsnLen& bytesDecoded, ENV_TYPE env);
    void FillBitStringStk (BUF_TYPE b, AsnLen elmtLen0,
                           AsnLen& bytesDecoded, ENV_TYPE env);

};
```

The `AsnBits` class contains a pointer to the bits and an integer that holds the length in bits of the BIT STRING.

In addition to the standard methods, the `AsnBits` class has methods for initializing and comparing bit string values and methods for setting and getting individual bits in a value.

An `AsnBits` value can created three ways: from the number of bits needs, from a `char*` and its bit length or from another `AsnBits` value. Look at the constructors and the `Set` and `ReSet` methods.

`SetBit` and `ClrBit` can be used for setting the values of individual bits in the BIT STRING value. Given the bit's index, `SetBits` sets the that bit to one. `ClrBit` sets the bit of the given index to zero. The bit indexes start at zero, with zero being the first (most signficant) bit in the BIT STRING. `GetBit` will return TRUE if the specified bit is one and FALSE if the bit is zero. If the given bit index is too large, `SetBit` and `ClrBit` do nothing and `GetBit` returns FALSE.

The `==` and `!=` operators have been overloaded such that given two `AsnBits` values, they will behave as expected.

Each `AsnBits` value stores its bit string in a single contiguous block of memory. Received BIT STRING values that were encoded in the constructed form are converted to the simple, flat form (see Section 4.8). Snacc provides no facility for encoding or internally representing constructed BIT STRING values.

6.10 OCTET STRING

OCTET STRING values are represented with the `AsnOcts` class. From `snacc/c++_include/asn_octs.h`:

```
class AsnOcts : public AsnType
{
  protected:
    char* octs;
    unsigned long int octetLen;

  public:
    // constructor and Set alway copy strings so destructor can always delete
    AsnOcts(): octs(NULL), octetLen(0) { }
    AsnOcts(const char* str)  { Set(str); }
    AsnOcts(const char* str, const unsigned long int len) { Set(str,len); }
    AsnOcts(const AsnOcts& o) { Set(o); }
    ~AsnOcts() { delete octs; }

    AsnOcts& operator=(const AsnOcts& o) { ReSet(o); return *this; }
    AsnOcts& operator=(const char* str) { ReSet(str); return *this; }

    // these set the octs and octetLen values
    void Set(const char* str, unsigned long int len);
    void Set(const AsnOcts& o);
    void Set(const char* str);

    // these free the old octs value and
    // then reset the octs and octetLen values
    void ReSet(const char* str, unsigned long int len);
    void ReSet(const AsnOcts& o);
    void ReSet(const char* str);

    operator char*() { return octs;}
    unsigned long int Len() {return octetLen;}

    operator==(const AsnOcts& o)  { return(OctsEquiv(o)); }
```

```
    operator!=(const AsnOcts& o)  { return(!OctsEquiv(o)); }

    AsnLen BEnc( BUF_TYPE b);
    void BDec( BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env);

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

    int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded);
    int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded);

    void Print(ostream& os);
    AsnType* Clone() { return new AsnOcts; }

  private:
    int OctsEquiv(const AsnOcts& o);
    void FillBitStringStk (BUF_TYPE b, AsnLen elmtLen0,
                           AsnLen& bytesDecoded, ENV_TYPE env);
    void BDecConsOcts (BUF_TYPE b, AsnLen elmtLen,
                                 AsnLen& bytesDecoded,
                                 ENV_TYPE env);
};
```

The AsnOcts class contains a pointer to the octets and an integer that holds the length in octets of the OCTET STRING.

There are four constructors for AsnOcts. The parameterless constructor will initialize the octet string to zero length with a NULL octets pointer. The constructor that takes a single char* assumes that the given string is NULL terminated and initializes the octet pointer with a pointer to a copy of the given string and sets the octetLen to the strlen of the the string (this does not usually included the NULL terminator). The constructor that takes char* and a length, len, initializes the octets pointer to point to a copy of len characters from the given string and sets the octetLen to len. The last constructor will initialize an AsnOcts value by copying the given AsnOcts value.

As with the BIT STRING content decoder, OCTET STRING content decoder can handle constructed values. These are handled in the same way as the constructed BIT STRING values; they are converted to the simple contiguous representation. Every OCTET STRING value will automatically have a NULL terminator appended to it; this extra character will not be included in the string's length and will make some strings easier to deal with for printing etc.

The operator char*() is defined for the AsnOcts class to return a pointer to the octets. This may be useful for passing the octets to other functions such as strcpy etc. The Len method returns the length in bytes of the the string value.

The == and != operators have been overloaded such that given two AsnOcts values, they will behave as expected.

6.11 OBJECT IDENTIFIER

OBJECT IDENTIFIER values are represented with the `AsnOid` class. From `snacc/c++_include/asn_oid.h`:

```
class AsnOid: public AsnType
{
  protected:
    char* oid;
    unsigned long int octetLen;

  public:
    AsnOid(): oid(NULL), octetLen(0) {}
    AsnOid(const char* encOid, unsigned long int len) { Set(encOid, len); }
    AsnOid(const AsnOid& o) { Set(o); }
    AsnOid(unsigned long int a1, unsigned long int a2,
           long int a3 = -1, long int a4 = -1, long int a5 = -1,
           long int a6 = -1, long int a7 = -1, long int a8 = -1,
           long int a9 = -1, long int a10 = -1, long int a11 = -1);
    ~AsnOid() { delete oid; }

    AsnOid& operator=(const AsnOid& o) { ReSet(o); return *this;}

    // Set methods overwrite oid and octetLen values
    void Set(const char* encOid, const unsigned long int len);
    void Set(const AsnOid& o);

    // first two arc numbers are mandatory.  rest are optional
    // since negative arc nums are not allowed in the encodings,
    // use them to indicate the 'end of arc numbers' in the
    // optional parameters
    void Set(unsigned long int a1, unsigned long int a2,
        long int a3 = -1, long int a4 = -1, long int a5 = -1,
        long int a6 = -1, long int a7 = -1, long int a8 = -1,
        long int a9 = -1, long int a10 = -1, long int a11 = -1);


    // ReSet routines are like Set except the old oid value is freed
    void ReSet(const char* encOid, const unsigned long int len);
    void ReSet(const AsnOid& o);
    void ReSet(unsigned long int a1, unsigned long int a2,
        long int a3 = -1, long int a4 = -1, long int a5 = -1,
        long int a6 = -1, long int a7 = -1, long int a8 = -1,
        long int a9 = -1, long int a10 = -1, long int a11 = -1);

    operator char*() { return oid; }
    unsigned long int Len() {return octetLen;}
    unsigned long int AsnOid::NumArcs();

    operator==(AsnOid& o)  { return(OidEquiv(o)); }
```

```
    operator!=(AsnOid& o)  { return(!OidEquiv(o)); }

    AsnLen BEnc( BUF_TYPE b);
    void BDec( BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env);

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

    int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded);
    int BDecPdu( BUF_TYPE b, AsnLen& bytesDecoded);

    void Print(ostream& os);
    AsnType* Clone() { return new AsnOid; }

  private:
    int OidEquiv(AsnOid o);
};
```

The `AsnOid` stores OBJECT IDENTIFIER values in their encoded form to improve performance. It seems that the most common operation with OBJECT IDENTIFIERs is to compare for equality, for which the encoded representation (which is canonical) works well.

The `AsnOid` is very similar to the `AsnOcts` class in all respects, except that its content decoding routine does not need to handle constructed encodings.

The `AsnOid` class has four constructors which are similiar to those of the `AsnOcts` class. A special constructor that takes arc numbers as parameters and uses default parameters is provided. An OBJECT IDENTIFIER value must have at least two arc numbers so the first two parameters do not have default values. All of the other parameters are optional; since their default value of `-1` is an invalid arc number (they must be positive) they will not be used in the value. For example to build the value `{1 2 3}` you simply use `AsnOid(1,2,3)`. This constructor is convenient but is more expensive in terms of CPU time that the others.

The `operator char*()` is defined for the `AsnOid` class to return a pointer to the encoded OBJECT IDENTIFIER value. This may be useful for passing the octets to other functions such as `strcpy` etc. The `Len` method returns the length in bytes of the the encode OBJECT IDENTIFIER value (NOT the number arcs in the value). `NumArcs` returns the number of arcs that the value is comprised of.

The `==` and `!=` operators have been overloaded such that given two `AsnOcts` values, they will behave as expected.

6.12 SET OF and SEQUENCE OF

In the C ASN.1 library, the list type was in the library because it was generic and every SET OF and SEQUENCE OF was defined as an `AsnList`. In C++, a new class is defined every list, providing a type safe list mechanism. This is described in the previous chapter.

6.13 ANY and ANY DEFINED BY

The ANY DEFINED BY type can be handled automatically by snacc provided you use the SNMP OBJECT-TYPE macro to specify the identifier to type mappings. The identifier can be an INTEGER or OBJECT IDENTIFIER. Handling ANY types properly will require modifications to the generated code since there is no identifier associated with the type.

Look at the C and C++ ANY examples and the `any.asn1` file included with this release for information on using the OBJECT-TYPE macro. Note that the OBJECT-TYPE macro has been modified slightly to allow INTEGER values (identifiers).

An ANY DEFINED BY type is represented by the `AsnAny` class. The following is from the `snacc/c++_include/asn_any.h`.

```
/* AnyInfo is a hash table entry */
class AnyInfo
{
  public:
    int    anyId;  /* will be a value from the AnyId enum */
    AsnOid oid;    /* will be zero len/null if intId is valid */
    AsnInt intId;
    AsnType* typeToClone;
};

class AsnAny: public AsnType
{
  public:
    static Table* oidHashTbl;  // all AsnAny class instances
    static Table* intHashTbl;  // share these tables
    AnyInfo* ai; // points to entry in hash tbl for this type
    AsnType* value;

    AsnAny() { ai = NULL; value = NULL; }

    // class level methods
    static void InstallAnyByInt (AsnInt intId, int anyId, AsnType* type);
    static void InstallAnyByOid (AsnOid& oid,  int anyId, AsnType* type);

    int GetId() {return((ai)? ai->anyId: -1); }
    void SetTypeByInt(AsnInt id);
    void SetTypeByOid(AsnOid& id);

    AsnLen BEnc(BUF_TYPE b);
    void BDec(BUF_TYPE b, AsnLen& bytesDecoded, ENV_TYPE env);

    int BEncPdu( BUF_TYPE b, AsnLen& bytesEncoded);
```

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

    void Print(ostream& os) { value->Print(os); }
};
```

The C++ mechanism is similar to the C mechanism which uses hash tables to hold the identifier to type mappings. In this section we will discuss the main differences of the C++ ANY DEFINED BY handling mechanism. You should read Section 4.12 for caveats and other important information.

In C, the hash table entry held the size of the type and pointers to its encode, decode, free etc. routines to describe the type. In C++ these have been replaced with a pointer to an instance of the type. A hash table entry contains:

- the `anyId`
- the INTEGER or OBJECT IDENTIFIER that maps to it
- a pointer to an instance of the identified type

All C++ ASN.1 types use the `AsnType` base class which designates the following functions as virtual:

- `BDec`
- `BEnc`
- `Print`
- `Clone`

This allows the ANY DEFINED BY handling routines to treat any type as an `AsnType`. So, for each type the ANY DEFINED BY handling code has access to the four virtual methods. Note that the `value` field in the `AsnAny` class and the `typeToClone` field in the `AnyInfo` class are both `AsnType*`.

To build an ANY DEFINED BY value, simply set the value field in the `AsnAny` object to point to the object you want to encode. Then set the identifier field for that ANY DEFINED BY value to the correct identifier (as generated for its OBJECT-TYPE macro value). It is very important to do this correctly because the encoder will simply call the virtual `BEnc` routine for the object pointed to by the `AsnAny`'s value field. There is not attempt to make sure that the identifier field's value matches the object that was encoded.

A potential solution to the last identifier problem is to add a type id field to the `AsnType` base class. Snacc could generate a unique identifier (that would be stored in the base class) for each type. The encoder could then check identifiers between the value being encoded and the value stored in the hash table. The identifier in the base class could easily be automatically set (correctly) from the constructors for each type (constructors are snacc generated). It would be difficult to ensure unique identifiers for each type between modules if the ASN.1 modules were compiled separately.

Before an ANY DEFINED BY value can be decoded, the field that contains its identifier must have been decoded and used with the AsnAny value's SetTypeByInt or SetTypeByOid methods. Then the ANY DEFINED BY value can be decoded by calling its (AsnAny) BDec routine. This in turn calls the Clone routine on the type in the hash table entry to generate the correct object. Then the BDec method of the newly created object is called.

When the C ANY DEFINED BY decoder allocates a value, it uses the size information for the identified type. This is not safe for C++ so the virtual Clone routine was added to the AsnType base class. This allows the proper constructor mechanism to be used when allocating the value.

The virtual Clone routine simply calls its type's parameterless constructor via new (hence every ASN.1 type's class must have a parameterless constructor). Clone is a poor name since the routine only produces a new instance of the given type without copying the original's data.

The hash tables are automatically initialized using the C++ constructor mechanism. You do not need to call any initialization routines as described in the C chapter.

6.14 Buffer Management

The C++ buffer management provided with snacc is similar to that of the C SBufs. The following is from snacc/c++_incluce/asn_buf.h:

```
class AsnBuf
{
  protected:
    char* dataStart;
    char* dataEnd;
    char* blkStart;
    char* blkEnd;
    char* readLoc;
    int writeError;
    int readError;

  public:
    void Init( char* data, unsigned long int dataLen);
    void ResetInReadMode();
    void ResetInWriteRvsMode();
    void InstallData(char* data, unsigned long int dataLen);
    unsigned long int DataLen();
    char* DataPtr();
    unsigned long int BlkLen();
    char* BlkPtr();
    int Eod();
```

```
    /* the following 9 methods are required  */
    int ReadError();
    int WriteError();
    void Skip(unsigned long int skipLen)
    unsigned long int Copy( char* dst, unsigned long int copyLen)
    unsigned char PeekByte()
    char* GetSeg(unsigned long int* lenPtr)
    void PutSegRvs( char* seg, unsigned long int segLen);
    unsigned char GetByte();
    void PutByteRvs(unsigned char byte);
};
```

This is the only buffer type provided with the C++ library.

6.15 Dynamic Memory Management

The built-in C++ memory management system is used by snacc (`new` and `delete`). Better performance might be gained by using an different management scheme.

To change `new` and `delete` to use your own memory management sheme the best way to start is by defining them as virtual in the `AsnType` base class. More information on providing your own memory management can be found in Stroustrup [8].

6.16 Error Management

The C++ ASN.1 error management is identical to that of the C ASN.1 model. C++ exception handling (`try` and `catch`) were not used to replace `setjmp` and `longjmp` because they were not implemented by the C++ compiler we used.

Chapter 7

Modifying the Compiler

The compiler consists of about 30000 lines of yacc, lex and C code (another 7000 for the runtime library routines). The best way to understand the compiler internals is to understand the module data structure (`snacc/src/asn1module.h`) and to read the compiler chapter in this document to gain a conceptual understanding of each pass of the compiler.

The most common form of modification will likely be for macro handling. To understand this, look at the way the OBJECT-TYPE macro is treated in:

asn1.lex add any new keywords

asn1.yacc parse the macro into the desired data structure. Use the existing productions as much as possible.

link_type.c link any type defined or referenced in the macro

link_values.c link any value defined or referenced in the macro

do_macros.c perform any semantic action for the macro

normalize.c move any type and value definitions in the macro to the top level so the code generator can generate code for them (without looking in the macro).

code generators to convert any special semantics into useful C or C++. This phase is likely to be dependent on the generated code's target environment.

Chapter 8

Future Work

There are still many interesting and useful things that can be done to snacc. To name a few, snacc could be improved to:

- support the new features of 1992 ASN.1
- generate interpreted decoders and encoders
- generate forward encoders that use only the indefinite length form for constructed BER values
- support new encoding rules such as the variants of Packed Encoding Rules (PER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).
- parse complex values.

1992 ASN.1 has many improvements such as eliminating macros and adding stronger mechanisms for adding type semantics. With stronger semantics, the compiler can do more for the user.

Interpreted encoders and decoders work by interpreting a description of the type to be encoded or decoded and acting accordingly. Instead of generating C or C++ code, the compiler produces a simple description of each type. While potentially slower than compiled encoders and decoders, the interpreted versions would likely take up less memory and could be easily re-configured during runtime. The internal representation (non-encoded) of a value presents and interesting problem (remember no abstract syntax specific .h files are available).

New encoding rules such as PER offer improved compression and possibly better performance in certain cases. They can be added to the compiler by adding a new backend to the compiler and providing a runtime library.

The ability to parse complex ASN.1 value notation values is useful for protocol testing. It allows PDUs to be defined in a human readable form and converted into their encoded representation. The combination of an interpreted encoder and decoder with complex value parsing and support for newer encoding rules would very useful for protocol testing.

Appendix A

The Module Data Structure ASN.1 Definition

The ASN.1 modules are parsed into an internal data structure. The data structure was initially defined in ASN.1 with the idea that if we needed to write a parsed module to disk, the ASN.1 encoding routines could be used. No file format was needed so the ASN.1 was merely and exercise.

This procedure highlighted the problem with circular links and index like links in ASN.1 data structures. BER does not support this type of linking; to handle it, the offending links can be made optional and not encoded. After decoding, these links need to be re-established.

The following is the ASN.1 definition of the Module data structure. The C translation (as generated by an early version of snacc) can be found in `snacc/src/asn1module.h`.

```
--
-- This module describes the data structure used to represent the
-- compiled ASN.1.
-- Using ASN.1 for the internal data structure allows writing
-- (encoding) to disk for storage (not done yet due to recursive
--  refs back to the module)
--
-- Mike Sample 91/08/29
-- Modifed 92/05  MS
--

Asn1Module DEFINITIONS IMPLICIT TAGS ::=
BEGIN

-- exports everything
-- imports nothing


Modules ::= [APPLICATION 0] IMPLICIT SEQUENCE
```

```
{
    creationTime INTEGER,
    modules ModuleList
}

ModuleList ::= SEQUENCE OF Module


Module ::= SEQUENCE
{
    status ENUMERATED { mod-ok(0), mod-not-linked(1), mod-error(2) },
    modId      ModuleId,
    tagDefault ENUMERATED { explicit-tags(0), implicit-tags(1) },
    exportStatus  ENUMERATED { exports-all(0), exports-nothing(1),
                               exports-some(2) },
    imports    ImportModuleList,
    typeDefs   TypeDefList,
    valueDefs  ValueDefList,
    hasAnys    BOOLEAN,

    asn1SrcFileName MyString,
    cHdrFileName    MyString,
    cSrcFileName    MyString,
    cppHdrFileName  MyString,
    cppSrcFileName  MyString
}


ModuleId ::= SEQUENCE
{
    name MyString,
    oid  OBJECT IDENTIFIER OPTIONAL --snacc cTypeName:"OID" isPtr:"TRUE"
}


ImportModuleList ::= SEQUENCE OF ImportModule


ImportModule ::= SEQUENCE
{
    modId       ModuleId,
    importElmts ImportElmtList,
    moduleRef   Module --snacc isEncDec:"FALSE"
}


ImportElmtList ::= SEQUENCE OF ImportElmt


ImportElmt ::= SEQUENCE
{
```

```
    resolvedRef CHOICE
    {
        type  [0] TypeDef,  -- not encoded
        value [1] ValueDef  -- not encoded
     } OPTIONAL,
    name MyString,
    privateScope BOOLEAN -- true if from MODNAME.TYPE ref
}

TypeDefList ::= SEQUENCE OF TypeDef

OidOrInt ::= CHOICE
{
    oid OBJECT IDENTIFIER,
    intId INTEGER
}

AnyRef ::= SEQUENCE
{
    anyIdName MyString,
    id OidOrInt
}

AnyRefList ::= SEQUENCE OF AnyRef

TypeDef ::= SEQUENCE
{
    exported     BOOLEAN,
    recursive    BOOLEAN,
    isPdu    BOOLEAN,
    localRefCount     INTEGER,
    importRefCount     INTEGER,
    visited      BOOLEAN,
    definedName  MyString,
    type         Type,
    cTypeDefInfo CTDI,
    cppTypeDefInfo CppTDI,
    attrList     AttributeList,
    refList      TypeDefList,
    anyRefs      AnyRefList
}


Tag ::= SEQUENCE
{
    tclass INTEGER, -- swap this for the BER_CLASS enum from basetypes.h
    form  INTEGER,  -- swap this for the BER_FORM enum
    code  INTEGER,
    explicit BOOLEAN,
    valueRef Value
}
```

```
Type ::= SEQUENCE
{
    optional   BOOLEAN,
    implicit   BOOLEAN,
    tags       TagList,
    defaultVal [0] IMPLICIT NamedValue OPTIONAL,
    subtypes   [1] Subtype OPTIONAL,
    basicType  [2] BasicType,
    lineNo     INTEGER,
    cTypeRefInfo CTRI,
    cppTypeRefInfo CppTRI,
    attrList   AttributeList
}

TagList ::= SEQUENCE OF Tag

AttributeList ::= SEQUENCE OF MyString

NamedNumberList ::= ValueDefList


-- BasicTypes with NULL need no more info that which type it is
-- (this is known from the choice id)

BasicType ::= CHOICE
{
    unknown         [0]  IMPLICIT NULL,
    boolean         [1]  IMPLICIT NULL,
    integer         [2]  IMPLICIT NamedNumberList,
    bitString       [3]  IMPLICIT NamedNumberList,
    octetString     [4]  IMPLICIT NULL,
    null            [5]  IMPLICIT NULL,
    oid             [6]  IMPLICIT NULL,
    real            [7]  IMPLICIT NULL,
    enumerated      [8]  IMPLICIT NamedNumberList,
    sequence        [9]  IMPLICIT NamedTypeList,
    sequenceOf      [10] IMPLICIT Type,
    set             [11] IMPLICIT NamedTypeList,
    setOf           [12] IMPLICIT Type,
    choice          [13] IMPLICIT NamedTypeList,
    selection       [14] IMPLICIT SelectionType,
    componentsOf    [15] IMPLICIT Type, --  [Resolved](local/import) type ref
    any             [16] IMPLICIT NULL,
    anyDefinedBy    [17] IMPLICIT AnyDefinedByType,
    localTypeRef    [19] IMPLICIT TypeRef,
    importTypeRef   [20] IMPLICIT TypeRef,
    macroType       [21] MacroType,
    macroDef        [22] IMPLICIT MacroDef --snacc isPtr:"FALSE"
}
```

```
MacroDef ::= MyString -- just keep the text for now

MacroType ::= CHOICE
{
    rosOperation    [0] IMPLICIT RosOperationMacroType,
    rosError        [1] IMPLICIT RosErrorMacroType,
    rosBind         [2] IMPLICIT RosBindMacroType,
    rosUnbind       [3] IMPLICIT RosBindMacroType,
    rosAse          [4] IMPLICIT RosAseMacroType,
    rosAc           [5] IMPLICIT RosAcMacroType,
    mtsasExtension  [6] IMPLICIT MtsasExtensionMacroType,
    mtsasExtensions [7] IMPLICIT MtsasExtensionsMacroType,
    mtsasExtensionAttribute [8] IMPLICIT MtsasExtensionAttributeMacroType,
    mtsasToken      [9] IMPLICIT MtsasTokenMacroType,
    mtsasTokenData [10] IMPLICIT MtsasTokenDataMacroType,
    mtsasSecurityCategory [11] IMPLICIT MtsasSecurityCategoryMacroType,
    asnObject       [12] IMPLICIT AsnObjectMacroType,
    asnPort         [13] IMPLICIT AsnPortMacroType,
    asnRefine       [14] IMPLICIT AsnRefineMacroType,
    asnAbstractBind      [15] IMPLICIT AsnAbstractBindMacroType,
    asnAbstractUnbind    [16] IMPLICIT AsnAbstractBindMacroType,
    asnAbstractOperation [17] IMPLICIT RosOperationMacroType,
    asnAbstractError     [18] IMPLICIT RosErrorMacroType,
    afAlgorithm     [19] IMPLICIT Type,
    afEncrypted     [20] IMPLICIT Type,
    afProtected     [21] IMPLICIT Type,
    afSignature     [22] IMPLICIT Type,
    afSigned        [23] IMPLICIT Type,
    snmpObjectType  [24] IMPLICIT SnmpObjectTypeMacroType
}


AnyDefinedByType ::= SEQUENCE
{
    fieldName MyString,   -- name of field that its defined by
    link NamedType OPTIONAL  -- REFERENCE not encoded
}


SelectionType ::= SEQUENCE
{
    fieldName MyString,   -- name of field in choice
    typeRef   Type,      -- [Resolved](local/import) type ref
    link NamedType OPTIONAL -- REFERENCE not encoded
}

NamedTypeList ::= SEQUENCE OF NamedType

NamedType ::= SEQUENCE
{
    fieldName  MyString, -- may be empty or NULL str
```

```
    type       Type
}


TypeRef ::= SEQUENCE
{
    typeName    MyString,
    moduleName  MyString, -- used for "modname.type" refs(may be null)
    module      Module,   --snacc isEncDec:"FALSE"
    link        TypeDef   --snacc isEncDec:"FALSE"
}



RosOperationMacroType ::= SEQUENCE
{
    arguments NamedType,
    result    NamedType,
    errors    [0] IMPLICIT TypeOrValueList OPTIONAL,
    linkedOps [1] IMPLICIT TypeOrValueList OPTIONAL
}

ValueList ::= SEQUENCE OF Value

TypeOrValueList ::= SEQUENCE OF TypeOrValue

TypeOrValue ::= CHOICE
{
    type  [0] IMPLICIT Type,
    value [1] IMPLICIT Value
}

OidList ::= SEQUENCE OF OBJECT IDENTIFIER


RosErrorMacroType ::= SEQUENCE
{
    parameter NamedType
}

RosBindMacroType ::= SEQUENCE
{
    argument NamedType,
    result   NamedType,
    error    NamedType
}


RosAseMacroType  ::= SEQUENCE
{
    operations       ValueList,
```

```
    consumerInvokes  ValueList,
    supplierInvokes  ValueList
}

RosAcMacroType ::= SEQUENCE
{
    nonRoElements       ValueList,
    bindMacroType       Type,
    unbindMacroType     Type,
    remoteOperations    Value,
    operationsOf        ValueList,
    initiatorConsumerOf ValueList,
    responderConsumerOf ValueList,
    abstractSyntaxes    OidList
}


MtsasExtensionMacroType ::=  SEQUENCE
{
    elmtType              [0] IMPLICIT NamedType OPTIONAL,
    defaultValue          [1] IMPLICIT Value    OPTIONAL,
    criticalForSubmission [2] IMPLICIT BOOLEAN  OPTIONAL,
    criticalForTransfer   [3] IMPLICIT BOOLEAN  OPTIONAL,
    criticalForDelivery   [4] IMPLICIT BOOLEAN  OPTIONAL
}


MtsasExtensionsMacroType ::= SEQUENCE
{
    extensions ValueList
}

MtsasExtensionAttributeMacroType ::= SEQUENCE
{
    type Type OPTIONAL
}

MtsasTokenMacroType ::= SEQUENCE
{
    type Type OPTIONAL
}

MtsasTokenDataMacroType ::= SEQUENCE
{
    type Type OPTIONAL
}

MtsasSecurityCategoryMacroType ::= SEQUENCE
{
    type Type OPTIONAL
}
```

```
AsnObjectMacroType ::= SEQUENCE
{
   ports  AsnPortList OPTIONAL
}

AsnPortList ::= SEQUENCE OF AsnPort

AsnPort ::= SEQUENCE
{
    portValue Value,
    portType  ENUMERATED
    {
         consumer-port(0),
         supplier-port(1),
         symmetric-port(2)
    }
}


AsnPortMacroType ::= SEQUENCE
{
    abstractOps     [0] IMPLICIT TypeOrValueList OPTIONAL,
    consumerInvokes [1] IMPLICIT TypeOrValueList OPTIONAL,
    supplierInvokes [2] IMPLICIT TypeOrValueList OPTIONAL
}


AsnRefineMacroType ::= INTEGER

AsnAbstractBindMacroType ::= SEQUENCE
{
    ports [0] IMPLICIT AsnPortList OPTIONAL,
    type  [1] IMPLICIT Type OPTIONAL
}


SnmpObjectTypeMacroType ::= SEQUENCE
{
   syntax Type,
   access ENUMERATED
       { snmp-read-only(0), snmp-read-write(1),
         snmp-write-only(2), snmp-not-accessible(3)},
   status ENUMERATED
       { snmp-mandatory(0), snmp-optional(1),
         snmp-obsolete(2), snmp-deprecated(3)},
   description [0] IMPLICIT Value OPTIONAL,
   reference   [1] IMPLICIT Value OPTIONAL,
   index       [2] IMPLICIT TypeOrValueList OPTIONAL,
   defVal      [3] IMPLICIT Value OPTIONAL
}
```

```
Subtype ::= CHOICE
{
    single [0] SubtypeValue,
    and    [1] IMPLICIT SubtypeList,
    or     [2] IMPLICIT SubtypeList,
    not    [3] Subtype
}

SubtypeList ::= SEQUENCE OF Subtype

SubtypeValue ::= CHOICE
{
    singleValue       [0] IMPLICIT Value,
    contained         [1] IMPLICIT Type,
    valueRange        [2] IMPLICIT ValueRangeSubtype,
    permittedAlphabet [3] Subtype, -- only valuerange or singleval
    sizeConstraint    [4] Subtype, -- only single value ints or val range
    innerSubtype      [5] IMPLICIT InnerSubtype
}


ValueRangeSubtype ::= SEQUENCE
{
    lowerEndInclusive BOOLEAN,
    upperEndInclusive BOOLEAN,
    lowerEndValue Value,
    upperEndValue Value
}


InnerSubtype ::= SEQUENCE
{
    constraintType ENUMERATED { full-ct(0), partial-ct(1), single-ct(2) },
    constraints ConstraintList
}

ConstraintList ::= SEQUENCE OF Constraint

Constraint ::= SEQUENCE
{
    fieldRef MyString,  -- not used if in single-ct, may be null
    presenceConstraint ENUMERATED
    {
        present-ct(0),
        absent-ct(1),
        empty-ct(2),
        optional-ct(3)
    },
    valueConstraints Subtype
```

```
}


ValueDefList ::= SEQUENCE OF ValueDef


ValueDef ::= SEQUENCE
{
   exported    BOOLEAN,
   definedName MyString,
   value       Value
}

Value ::= SEQUENCE
{
    type    Type OPTIONAL,
    valueType  INTEGER,  -- holds one of choiceId's def'd for BasicType
    basicValue BasicValue,
    lineNo     INTEGER
}

BasicValue ::= CHOICE
{
    unknown          [0]  IMPLICIT NULL,
    empty            [1]  IMPLICIT NULL,
    integer          [2]  IMPLICIT INTEGER,
    specialInteger   [3]  IMPLICIT SpecialIntegerValue,
    longInteger      [4]  IMPLICIT INTEGER,  -- put LONG before INTGEGER
    boolean          [5]  IMPLICIT BOOLEAN,
    real             [6]  IMPLICIT REAL,
    specialReal      [7]  IMPLICIT SpecialRealValue,
    asciiText        [8]  IMPLICIT OCTET STRING,
    asciiHex         [9]  IMPLICIT OCTET STRING,
    asciiBitString   [10] IMPLICIT OCTET STRING,
    oid              [11] IMPLICIT OBJECT IDENTIFIER,
    linkedOid        [12] IMPLICIT OBJECT IDENTIFIER, --snacc cTypeName:"OID"
    berValue         [13] IMPLICIT OCTET STRING,
    perValue         [14] IMPLICIT OCTET STRING,
    namedValue       [15] IMPLICIT NamedValue,
    null             [16] IMPLICIT NULL,
    localValueRef    [17] IMPLICIT ValueRef,
    importValueRef   [18] IMPLICIT ValueRef,
    valueNotation    [19] IMPLICIT OCTET STRING
}



SpecialIntegerValue ::= ENUMERATED { min-int(0), max-int(1) }
SpecialRealValue ::= ENUMERATED { minus-infinity-real(0), plus-infinity-real(1) }
```

```
ValueRef ::= SEQUENCE
{
    valueName  MyString,
    moduleName MyString, -- used for "modname.value" refs (may be null)
    link       ValueDef, --snacc isEncDec:"FALSE"
    module     Module    --snacc isEncDec:"FALSE"
}

NamedValue ::= SEQUENCE
{
   fieldName MyString, -- may be null
   value     Value
}

NamedValueList ::= SEQUENCE OF NamedValue

CTypeId  ::= ENUMERATED { c-choice(0), c-list(1), c-any(2), c-anydefinedby(3),
                          c-lib(4), c-struct(5), c-typeref(6), c-no-type(7),
                         c-typedef(8) }

-- C Type Def Info - info used for routine naming
-- and referencing from other types
CTDI ::= SEQUENCE
{
    asn1TypeId         INTEGER, --snacc cTypeName:"enum BasicTypeChoiceId"
    cTypeId            CTypeId,
    cTypeName          MyString,
    isPtrForTypeDef    BOOLEAN,
    isPtrForTypeRef    BOOLEAN,
    isPtrInChoice      BOOLEAN,
    isPtrForOpt        BOOLEAN,

               -- defines these names, used by references
    optTestRoutineName MyString, -- routine/macro to check whether
                                 -- opt type is present
    defaultFieldName   MyString, -- base for generating field names

    printRoutineName   MyString,
    encodeRoutineName  MyString,
    decodeRoutineName  MyString,
    freeRoutineName    MyString,
    isEncDec           BOOLEAN, -- if false, no routines are gen
                                -- and not included in encodings
    genPrintRoutine    BOOLEAN,
    genEncodeRoutine   BOOLEAN,
    genDecodeRoutine   BOOLEAN,
    genFreeRoutine     BOOLEAN,
    genTypeDef         BOOLEAN
}
```

```
--
-- CTRI (C Type Ref Info) is used for generating C typedefinitions
-- from the ASN.1 types info
CTRI ::= SEQUENCE
{
    cTypeId CTypeId,
    cFieldName MyString,
    cTypeName  MyString,
    isPtr BOOLEAN,
    isEndCType BOOLEAN,        -- false for struct/union def
    cNamedElmts CNamedElmts OPTIONAL,      -- for C_LIB bits/int/enums
    choiceIdValue INTEGER,     -- enum value of this c field
    choiceIdSymbol MyString,   -- this fields sym in choiceId enum
    choiceIdEnumName MyString,
    choiceIdEnumFieldName MyString,
    optTestRoutineName MyString, -- these names are gained from refd type def
    printRoutineName   MyString,   -- or are over-ridden snacc attribute comment
    encodeRoutineName  MyString,
    decodeRoutineName  MyString,
    freeRoutineName    MyString,
    isEncDec           BOOLEAN -- whether part of enc value
}

CNamedElmts ::= SEQUENCE OF CNamedElmt

CNamedElmt ::= SEQUENCE
{
    name MyString,
    value INTEGER
}


CppTDI ::= SEQUENCE
{
    asn1TypeId         INTEGER, --snacc cTypeName:"enum BasicTypeChoiceId"
    className          MyString,
    isEnc              BOOLEAN,
    isPtrForTypeDef    BOOLEAN,
    isPtrForTypeRef    BOOLEAN,
    isPtrInChoice      BOOLEAN,
    isPtrForOpt        BOOLEAN,
    optTestRoutineName MyString,
    defaultFieldName   MyString -- base for generating field names
}



CppTRI ::= SEQUENCE
{
    isEnc BOOLEAN,
    className MyString,
```

```
    fieldName MyString,
    isPtr BOOLEAN,
    namedElmts CNamedElmts,
    choiceIdSymbol MyString,
    choiceIdValue INTEGER,
    optTestRoutineName MyString
}

-- use snacc compiler directives to overide the builtin types.
--
-- All strings used in module data struct are null terminated so
-- can just use a char*
-- Note the snacc comments before the PrintableString
-- bind with the MyString TypeDef and the ones after PrintableString
-- bind with the PrintableString Type ref.


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

END
```

Bibliography

[1] CCITT. *Data Communications Networks Open systems Interconnection (OSI) Model and Notation, Service Defintion*, volume VIII of *Blue Book*, chapter Recommendation X.208, Specification of Abstract Syntax Notation One (ASN.1), pages 57–130. Omnicom, 115 Park St., S.E., Vienna, VA 22180 USA, November 1989.

[2] CCITT. *Data Communications Networks Open systems Interconnection (OSI) Model and Notation, Service Defintion*, volume VIII of *Blue Book*, chapter Recommendation X.209, Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), pages 130–151. Omnicom, 115 Park St., S.E., Vienna, VA 22180 USA, November 1989.

[3] Motorola Inc. *MC68881 Floating-Point Coprocessor User's Manual*. Motorola Inc., 1985.

[4] Gerald Neufeld and Son Vuong. An overview of asn.1. *IEEE Networks and ISDN Systems*, 23(5):393–415, Feb 1992.

[5] Gerald Neufeld and Yeuli Yang. An asn.1 to c compiler. *IEEE Transactions on Software Engineering*, 16(10):1209–1220, Oct 1990.

[6] Marshall T. Rose. *ISODE, The ISO Development Environment: User Manual*. Wollongong Group, 1129 San Antonio Rd. Palo Alto, California, USA, February 1990.

[7] Douglass Steedman. *ASN.1, The Tutorial and Reference*. Number ISBN 1871802067. Technology Appraisals Ltd., 1990.

[8] Bjarne Stroustrup. *The C++ Programming Language, 2nd Edition*. Number ISBN 0201539926. Addison-Wesley Publishing Co., 1991.