# Snacc 1.2rj: A High Performance ASN.1 to C/C++/IDL Compiler

Michael Sample
msample@cs.ubc.ca

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

augmented by:
Robert Joop
<rj@rainbow.in-berlin.de>

msample: February 1993, updated July 1993
rj: 1994/1995

*Preliminary documentation as of September 22, 1995*
*for Snacc 1.2rj.9*

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

# List of Figures

# Chapter 1

# Introduction

Snacc compiles ASN.1 [1] (Abstract Syntax Notation One) modules into C, C++, CORBA IDL [8] or type tables. 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".

This is release 1.2rj[1] of Snacc. This chapter lists only the differences to the original author's last release, Snacc 1.1. The list in this chapter is incomplete—for a more thorough enumeration, see the file `.../ChangeLog`.

New features include:

- The output files generated get names derived from their input file's name, with only the suffix replaced. This eases makefile writing, as now you can use simple suffix rules or other forms of filename pattern matching. The old behaviour, where the output files got their name from the ASN.1 module name, can be retained by using the `-mm` command line switch to `snacc`.
- The C++ backend generates code with a much more complete set of constructors, destructors and assignment operators.
- The C++ backend can supply the generated C++ classes with *meta* information about their own structure. This information can be used to build interpreted interfaces; the Snacc 1.2rj distribution contains a Tcl interface that uses this meta information as well as a Tcl script (that uses the Tcl interface) for a simple editor.
- Snacc has successfully been ported to Linux and Alpha OSF/1, and should be both byte order and 64 bit clean.
- A new backend that generates CORBA IDL (work-in-progress, not even alpha)

The makefiles have been rewritten. The old ones removed the `.o` files after successful compilation, and thus, for every tiny code change, a full recompilation took place!

[1] *1.2* since it is the successor of 1.1 and *rj* as i don't think that I'm the only one who worked on Snacc.

With the new set of makefiles, only those files that need to be remade are. Following usual conventions, the phony targets depend, check, install, clean and clobber have been added.
NOTE: the makefiles still are not safe for parallel making.
See Appendix E for some explanations of some the makefile tricks.
If you've got problems with the makefiles, please refer to the appendix!

The 1.1 release used five config.h files, and three almost identical copies of the ASN.1 C library. They have all been merged in the file `.../snacc.h`. A very small number (currently three) of compilation switches has been put into `.../policy.h`.

The previous release produced huge virtual inline functions. Due to their size, these inlines wouldn't get inlined anyway. Virtual functions don't get inlined (they get referenced via pointer in the virtual function table). Due to their size they wouldn't offer any speed advantage (the function call overhead diminishes). Instead, the compiler generated static functions in every `.C` file were the `.h` file is included! This inflates the `.o` files and executables real quick (I'm speaking of MBytes per executable). These functions have been turned into normal functions.

## 1.1 Configuring and Installing Snacc

First of all, if you haven't already done so, un-archive Snacc to produce the directory `snacc-1.2rj.`*`patchlevel`* and its contents. The directory `snacc-1.2rj.`*`patchlevel`* will henceforth be referred to as "`...`". The following tools are required to compile Snacc:

- `make` (GNU `make` is recommended)
- `patch` (for a tiny patch in `.../c-lib/`)
- `makedepend` or a look-alike (all of them have their advantages and disadvantages, it is hard to recommend any of them, see below)
- `lex` or GNU's `flex` (`flex` is recommended)
- `yacc` or GNU's `bison` (`bison` is recommended)
- a C compiler (it doesn't have to support ANSI, K&R will do)

Some versions of `yacc` may choke due to the large size of the `parse-asn1.y` file, however, we 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 `lex-asn1.l` 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-asn1.c` file to at least 2048. `Flex` does not seem to have this problem.

The configuration process has been simplified (at least for the installer of Snacc ;-) by the use of GNU autoconf.

The only file has may have to be edited is `.../policy.h`. It contains a few compilation switches you may want to toggle.

The behaviour of makedepend has been changed from X11R5 to R6. The new version keeps the source files' dirname and replaces the suffix only, the old version removed the dirname. The makefiles expect the new behaviour. If you've still got the R5 makedepend, the compiler's dependencies will be useless. (If you only install the code and don't make any source code changes, this won't hurt you.) If you haven't got `makedepend`, you can use any of the look-alikes, which often are `sh`-scripts calling the compiler with the `-M`-switch. If you don't plan to make any source code changes, you can replace `makedepend` with `/bin/true`.
Warning: MIT X11's makedepend is broken, in both R5 and R6. It silently does not produce any output for many of Snacc's C++ files (in `.../c++-lib/`).
The C compiler called with the `-M`-switch gives much better results, but is *much* slower.

The Snacc compiler and library C code has been written to support ANSI or non-ANSI C. The configuration script tries to find out whether your C compiler understands ANSI C.

The configuration script generates two files:

`.../makehead` gets included by all makefiles. It contains a lot of definitions used by make.

`.../config.h` contains all the machine, operating system, compiler and environment dependent settings. It is included by `.../snacc.h`.

The C++ runtime library is known to compile with both `gcc 2.5.8` and `gcc 2.6.3`. The latter has the bool type built-in (which the configuration script automatically detects).

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

```
%1 cd snacc-1.2rj.*
%2 ./configure
%3 make
```

If you wish to install only the C (including type tables) or only the C++ versions of the library, type `make c` or `make c++`, respectively, instead of `make`. If the make succeeds, the `snacc` binary should be present as `.../compiler/snacc`, the C runtime libraries, `libasn1csbuf.a`, `libasn1cebuf.a`, `libasn1cmbuf.a` and `libasn1ctbl.a`, should be in `.../c-lib/` and the C++ runtime library,

`libasn1c++.a` (and, if you compiled with the Tcl option enabled, another run-time library, `libasn1tcl.a`), should be in `.../c++-lib/`. The type table tools, `ptbl`, `pval` and `mkchdr`, will be in their respective directories under `.../tbl-tools/`.

After compiling the libraries, you can test the library routines by calling `make check` (or by calling `make c-check` or `make c++-check` to test the C or C++ library routines only, respectively).

Manual pages that contain information on running `snacc` and the type table tools can be found in `.../doc/`.

To install Snacc, you can call `make install` (or `make c-install` or `make c++-install`, respectively). This installs the snacc compiler binary, the libraries, the `.h` and `.asn1` files, the type table tools, as well as the manual pages into the usual directories.

To remove the `.o` and other intermediate files, you can call `make clean`. To remove the binaries, libraries and all other generated files as well, call `make clobber`.

## 1.2 Running Snacc

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

```
snacc [-h] [-P] [-t] [-e] [-d] [-p] [-f]
      [ -c | -C | -idl | -T <table output file>]
      [-meta <module>.<type>[,...]] [-mA | -mC]
      [-tcl <module>.<type>[,...]]
      [-u <useful types ASN.1 file>]
      [-mm] [-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. Alternatively, snacc can produce type tables that can be used for table based/interpreted encoding and decoding. The type table based methods tend to be slower than their C or C++ counterparts but they tend use less memory (table size vs. C/C++ object code).

Snacc may also be used to generate CORBA IDL. This part of Snacc is very new and I would rate it as pre-alpha.

The `-meta`, `-mA`, `-mC` and `-tcl` options are only present when the Tcl and Tk libraries where found at configuration time.

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 (printed to stderr).

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. If the target language is CORBA IDL, snacc will generate an `.idl` file for each module. The generated file names will be derived from the module's filenames, or from the module names if the `-mm` command line switch has been given.

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 the `-c`, `-C`, `-idl` or `-T` options should be specified.

**–C** causes snacc to generate C++ source code.

**–idl** causes snacc to generate CORBA IDL source code.

**–T *file*** causes snacc to generate type tables and write them to the given file *file*.

**–meta *types*** causes snacc to generate C++ classes with type meta information. Requires C++ functionality and therefore implies `-C` (C++ code generation).

The *types* denote the PDUs and have the following syntax: a comma-separated list of pairs of: module name, a dot, and a type name from that module. (Example: `snacc -tcl M1.T-a,M-2.Tb mod1.asn1 m2.asn1`)

**–mA** and **–mC** causes the metacode to use identifiers as defined in the ASN.1 source files or as used in the generated C++ code, respectively. (It defaults to `-mC`.)

**–tcl *types*** causes snacc to generate functions for a Tcl interface. Needs the type meta information and thus implies `-meta` (see above). The `-meta` option can and should be omitted, the *types* are as for the `-meta` option (the *types* arguments are additive, in case you specify both options).

**–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. These options do not affect type table generation. 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 `.../c-lib/inc/asn-config.h` to change the default memory system. For more information on the memory management see Section 5.14.

**–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 `.../asn1specs/asn-useful.asn1` and contains:

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

**–mm** This switch is supplied for backwards compatibility. Snacc versions 1.0 and 1.1 produced files with names generated from the ASN.1 module name contained in the input file. Snacc 1.2rj by default retains the input file name and replaces the suffix only. The new behaviour makes `makefile` writing easier, as with modern `make`s, pattern matching can be used.

**–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 were chosen to fit common C programming style. For all names, dashes in the ASN.1 source are converted to underscores. See Sections 4.2 and 6.2 for more naming information.

If the `-mm` switch has been given, 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 semicolons.

### 1.2.1 Known Bugs

- Snacc has problems with the following case:

```
Foo ::= SEQUENCE
{
  id IdType,
  val ANY DEFINED BY id
}
```

```
IdType ::= CHOICE
{
  a INTEGER,
  b OBJECT IDENTIFIER
}
```

The error checking pass will print an error to the effect that the id type must be INTEGER or OBJECT IDENTIFER. To fix this you must modify the error checking pass as well as the code generation pass. To be cheap about it, disable/fix the error checking and hand modify the generated code.

- The hashing code used for handling ANY DEFINED BY id to type mappings will encounter problems if the hash table goes more than four levels deep (I think this is unlikely). To fix this just add linear chaining at fourth level.
- The `.../configure` script should check whether the machine's floating point format is IEEE or whether the IEEE library exists.
- The C++ library severly lacks a convenient buffer management class that automatically expands like the C libraries' ExpBuf. What use is an efficient buffer management when you have got to build a loop a round snacc's encoding routine that reallocates larger buffers until the result fits?
- Where this document describes personal experiences, it is usually unclear to which author `I' refers. (One way to find out is to look at snacc 1.1's documentation.)

## 1.3 Reporting Bugs and Your Own Improvements

Snacc 1.1 was Michael Sample's final release. While he is watching Snacc's development, he isn't actively developing it himself.

Since there are quite a number of changes from release 1.1 to 1.2rj, bug reports and new features are best sent to me. I can be reached as `Robert Joop <rj@rainbow.in-berlin.de>` or `<rj@fokus.gmd.de>`.

# Chapter 2

# Introduction to Snacc Release 1.1

Snacc compiles ASN.1 [1] (Abstract Syntax Notation One) modules into C, C++ or type tables. 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".

This compiler was written so I could do some encoding performance research for my M.Sc. See [13], or [12] for the results of that research. A techreport will soo be available via ftp from UBC, in the same directory as snacc.

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 [14], [6], [1] and [2].

Compiling ASN.1 into C is not a new idea but many other tools such as UBC's CASN1 [7], ISODE's PEPY/POSY [11], 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.

The table driven encoders are useful for certain applications such as protocol testing. They are also useful if you need to dynamically load new ASN.1 definitions. It is also fairly simple to write your own special ASN.1 tools based on tables (e.g. a protocol tester that verifies that values conform to a given ASN.1 type definition). The price

of the flexibility is speed; they are slower ( 4 times) than the compiled C and C++ versions.

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

## 2.1 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/c_lib/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`. The type table library makefile uses `gcc` and is `snacc/tbl_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 all
```

If you wish to install only the C (including type tables) or only the C++ versions of the library, type `make c` or `make c++`, respectively, instead of `make all`. 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 type table library, *libasn1tbl.a* will be in `snacc/tbl_lib`. The type table tools, *ptbl*, *pval* and *mkchdr* will be in `snacc/bin`. 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.

## 2.2 Running snacc

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

```
snacc [-h] [-P] [-t] [-e] [-d] [-p] [-f]
```

```
[ -c | -C | -T <table output file>]
[-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. Alternatively, snacc can produce type tables that can be used for table based/interpreted encoding and decoding. The type table based methods tend to be slower than their C or C++ counterparts but they tend use less memory (table size vs. C/C++ object code).

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 (printed to `stderr`).

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 the `-c`, `-C` or `-T` options should be specified.

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

**-T *file*** causes snacc to generate type tables and write them to the given file *file*.

**-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. These options do not affect type table generation. 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 5.14.

**-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/asn-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 were chosen to fit common C programming style. For all names, dashes in the ASN.1 source are converted to underscores. See Sections 4.2 and 6.2 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 semicolons.

### 2.2.1 Known Bugs

Snacc has problems with the following case:

```
Foo ::= SEQUENCE
{
    id IdType,
    val ANY DEFINED BY id
}

IdType ::= CHOICE
{
    a INTEGER,
    b OBJECT IDENTIFIER
}
```

The error checking pass will print an error to the effect that the id type must be INTEGER or OBJECT IDENTIFER. To fix this you must modify the error checking pass as well as the code generation pass. To be cheap about it, disable/fix the error checking and hand modify the generated code.

The hashing code used for handling ANY DEFINED BY id to type mappings will encounter problems if the hash table goes more than four levels deep (I think this is unlikely). To fix this just add linear chaining at fourth level.

On the deficiency side of things, the C++ classes really need to have free methods defined for them. (Unless you have replaced new with something like nibble memory)

## 2.3 Reporting Bugs and Your Own Improvements

This (1.1) is the final release of snacc (I have finished my M.Sc). Gerald Neufeld <neufeld@cs.ubc.ca> was my supervisor but he does not have time to deal with support (it is all my code anyway). Luckily, a colleague has kindly offered to receive the bug reports and to coordinate work done by others (i.e. you). His name is Barry Brachman <brachman@cs.ubc.ca>. He did not write the code (35,000+ lines of C) but he has used snacc for X.500 work. He may be able to point you to someone who has fixed or encountered the same bug. Anyway, be nice to him, it's not his job.

Even though this is the second release of snacc, bugs are still likely. In fact, this release was quite rushed so there are probably lots of stupid installation bugs etc. If you find some bugs or have other comments, please send email to snacc-bugs@cs.ubc.ca (these will get to Barry and Gerald). Please include the offending ASN.1 source, the command line options you were using and the hardware and operating system configuration.

If you are really keen and hack in new goodies, please share. Send them to Barry or snacc-bugs@cs.ubc.ca. Look in snacc/README.future for things you could work on.

As I mentioned, I have entered the real world. I am now working with Open Systems Solutions (based in New Jersey). If your application needs a commercially developed and supported ASN.1 compiler, try calling 1-609-987-9073 (Yeah, I know this is a plug, but it's a good company).

# Chapter 3

# Compiler Design

## 3.1 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 (`.../compiler/core/parse-asn1.y`) 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++, IDL or type table

The source code for the compiler resides in `.../compiler/` and the back ends are in `.../compiler/back-ends/c-gen/`, `.../compiler/back-ends/c++-gen/` and `.../compiler/back-ends/idl-gen/`.

## 3.2 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 redefine the existing ones without modifying snacc. This also has the benefit that names of useful types are not keywords 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 3.4 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.

## 3.3 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`, `lex-asn1.l`, `parse-asn1.y` 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.

## 3.4 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 a linking error is reported. Note that this form of type reference provides a special scope that does not conflict with other local or imported types in that module.

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

For the remaining unresolved type references (failed local and legal import resolution and are not of the form *modulename.typename*), the linker looks in the useful types module, if 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.

## 3.5 Pass 4: Parsing Values

The fourth pass attempts to parse any value that 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 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 the OBJECT IDENTIFIER tree. Any arc value that is represented by a single identifier is checked to see if it is one of the arc names defined in the OBJECT IDENTIFIER tree; context is ignored. If the identifier matches one of the arc names then its value is set accordingly. The lack of context sensitivity in snacc's algorithm may cause the arc name to link with an arc name from the OBJECT IDENTIFIER tree when a local or imported INTEGER was desired. The following is the list special arc names that snacc understands and their values (see `.../compiler/core/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

## 3.6 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 3.4).

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

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

are equivalent to this:

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

Note that in version 1.0, named arcs (e.g. stimpy(3)) were promoted to full integer values. This was wrong—many standards re-used them (e.g. X.500 and ds(5)) leading to multiply defined integer values. If you want to improve the value parsing, look in `.../compiler/core/val-parser.c`

## 3.7 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 nor-

mal scope. This way, the code generator does not have to know about macros to generate code for the types defined within them.

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

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

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

The following macros are parsed:

- OPERATION (ROS)
- ERROR (ROS)
- BIND (ROS)
- UNBIND (ROS)
- APPLICATION-SERVICE-ELEMENT (ROS)
- APPLICATION-CONTEXT
- EXTENSION (MTSAS)
- EXTENSIONS (MTSAS)
- EXTENSION-ATTRIBUTE (MTSAS)
- TOKEN (MTSAS)
- TOKEN-DATA (MTSAS)
- SECURITY-CATEGORY (MTSAS)
- OBJECT (X.407)
- PORT (X.407)
- REFINE (X.407)
- ABSTRACT-BIND (X.407)
- ABSTRACT-UNBIND (X.407)
- ABSTRACT-OPERATION (X.407)
- ABSTRACT-ERROR (X.407)
- ALGORITHM (X.509)
- ENCRYPTED (X.509)
- PROTECTED (X.509)
- SIGNATURE (X.509)
- SIGNED (X.509)
- OBJECT-TYPE (SNMP)

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

## 3.8 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.

## 3.9 Pass 8: Marking Recursive Types

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

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

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

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

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

## 3.10 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).

## 3.11 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.

## 3.12 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 [7] 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).

## 3.13 Pass 12: Generating Code

This pass creates and fills the source files with C or C++ code or produces a type table containing the type descriptions from all of the parsed modules, including the useful types module (if given). 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 type table data structure is similar to snacc's parse tree for the ASN.1 modules but it is much simpler. This is because all of the type linking and error checking has been done. The type definitions in the type tables are in defined by the type sorting pass (dependency).

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

# Chapter 4

# C Code Generation

## 4.1 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 a 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, shortcuts that can be taken without sacrificing the robustness of code are used.

The generated code can be quite large; large reductions 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 `.../compiler/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 .../asn1specs/asn-useful.asn1 .../asn1specs/
ex1.asn1
```

This 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 and 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.

## 4.2 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 have default field names defined by the compiler (see `.../compiler/back-ends/c-gen/rules.c` and `.../compiler/back-ends/c++-gen/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.

## 4.3 ASN.1 to C Data Structure Translation

To handle the different scoping rules between ASN.1 and C, 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 5.

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.

## 4.4 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 en-

capsulates 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 4.9 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 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 an 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 5.13) for how to check a buffer for a write error.

## 4.5 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, AsnL
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 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 is 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 5.15 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 5.14) 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).

## 4.6 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 in the following style:

```
{ -- SEQUENCE --
    17,
    '436c696d6220617420537175616d697368'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.

## 4.7 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 5.14. 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.

## 4.8 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;
```

## 4.9 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 and the value is what the attribute's new value will be. The attribute value pairs can be listed in a single –snacc comment or spread out in 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. Let's 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, let's 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 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 wish 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 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 C function or macro name.

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

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

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

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

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

**isEncDec** If this type is used in a SET or SEQUENCE then the field 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.

**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.

## 4.10 Compiling the Generated C Code

The generated C code (and libraries) can be compiled by both ANSI and K&R C compilers. C function prototypes use the PROTO macro and C function declarations use the PARAMS macro. These macros are defined in `.../snacc.h` and their definitions depend on whether the __USE_ANSI_C__ flag has been defined in `.../config.h`.

When compiling the generated C code you will need:

1. The include directory where the files from `.../c-lib/inc/` have been installed into in your include path so the C sources can include the library header files. The header files should be included with statements like #include <snacc/c/asn-incl.h> and your C compiler should be supplied with `-I/usr/local/include` in case snacc got installed under `/usr/local/`.

2. to link with the correct C ASN.1 runtime library, depending on the buffer type you choose. In case snacc got installed under `/usr/local/`, your linker may need to be supplied with `-L/usr/local/lib` and one of `-lasn1cebuf`, `-lasn1cmbuf` or `-lasn1csbuf` as arguments.

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

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

# Chapter 5

# C ASN.1 Library

## 5.1 Overview

Each library type has a file in the `.../c-lib/src/` and `.../c-lib/inc/` 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 referred 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 `.../c-examples/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.

## 5.2 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 `.../c-lib/inc/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 is 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 it read past the end of the data when decoding the tag.

## 5.3 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 4.4. 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 encoded 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.

## 5.4 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 4.4 and 4.5, 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 initialization 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 one octet.

## 5.5 INTEGER

The INTEGER type is represented by a 32 bit integer type, AsnInt. The C integer type chosen depends on the machine and compiler and may be int, long or short, whatever is 32 bits in size. 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 int for a larger positive value range.

```
typedef int AsnInt;
typedef unsigned 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.

## 5.6 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);
```

## 5.7 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 of a 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.

## 5.8 BIT STRING

The BIT STRING type is represented by the AsnBits structure. It contains a pointer to the bits 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 iden-

tical. 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 performance loss) and will not shrink after growing. If you wish to modify this behaviour, change the `.../c-lib/inc/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.

## 5.9 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.

## 5.10 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)
```

## 5.11 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 a doubly linked list 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.

A 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 5.14). 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 type, 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.

## 5.12 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* if the SNMP OBJECT-TYPE [10] macro is used to specify the identifier value to type mappings. The identifier can be an INTEGER or OBJECT IDENTIFIER. Handling ANY types properly will require modifications to the generated code since there is no identifier associated with the type.

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

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

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

- the `anyId`
- the INTEGER or OBJECT IDENTIFIER that maps to it

- the size in bytes of the identified data type
- a pointer to the type's PDU encode routine
- a pointer to the type's PDU decode routine
- a pointer to the type's print routine
- a pointer to the type's free routine

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

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

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

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

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

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

When encoding, before the ANY DEFINED BY value is encoded, `SetAnyTypeByOid` or `SetAnyTypeByInt` (depending on the type of the identifier) is called with the current identifier value to set the `AsnAny` value's `ai` pointer to the proper hash table entry. Then to encode the ANY DEFINED BY value, the encode routine pointed to from the hash table entry is called with the `value void *` from the `AsnAny` value. The `value void *` in the `AsnAny` should point to a value of the correct type for the given identifier, if the user set it up correctly. Note

that setting the `void *` value is not type safe; one must make sure that the value's type is the same as indicated by the identifier.

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

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

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
```

## 5.13 Buffer Management

Encoding and decoding performance is heavily affected by the cost of writing to and reading from buffers, thus, efficient buffer management is necessary. Flexibility is also important to allow integration of the generated encoders and decoders into existing environments. To provide both of these features, the calls to the buffer routines are actually macros that can be configured as you want (see `.../c-lib/inc/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 means 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. I have implemented buffers like this for the table tools (performace is already hosed so slower buffer routines are a drop in the bucket). See the type tables section for their description.

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.

### 5.13.1 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 the 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.

### 5.13.2 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).

### 5.13.3 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 `.../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 5.13.1 and 5.13.2).

### 5.13.4 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 `.../c-lib/exp-buf.c` for an ASCII drawing of the ExpBuf buffers. Take a look a the `.../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 is reversed) 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 si 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 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 your 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 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 `.../c-lib/inc/asn-config.h`). These ExpBuf routines adhere to the buffer routine prototypes and semantics defined in Sections 5.13.1 and 5.13.2.

```
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);
```

### 5.13.5 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 on the size of the encodings you will be 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 `.../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.

### 5.13.6 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 advances 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 to the first byte of the BER value to be decoded. Look in `.../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.

### 5.13.7 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 a different 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.

## 5.14 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 librarys and the generated decode routines allocate memory using the Asn1Alloc routine. The runtime librarys 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 routines 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 a value's memory using a 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 if 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 too 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.

## 5.15 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 `.../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 `.../c-lib/inc/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 6

# C++ Code Generation

## 6.1 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 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 throw are obvious replacements for the setjmp and longjmp used by the C decoders. Unfortunately 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 `.../compiler/back-ends/c++-gen/` directory if you want to alter it.

## 6.2 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 `.../compiler/back-ends/c-gen/rules.c` and `.../compiler/back-ends/c++-gen/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 STRINGs 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.

## 6.3 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. it inherits from the AsnType base class
2. it has a parameterless constructor
3. it has a copy constructor
4. it has a destructor
5. it has a clone method, Clone
6. it has an assignment operator
7. it has a content encode and decode method, BEncContent and BDecContent
8. it has a PDU encode and decode method, BEnc and BDec
9. it has a top level interfaces to the PDU encode and decode methods (handles the setjmp etc.) for the user, BEncPdu and BDecPdu
10. it has a print method, Print, a virtual function that gets called from a global <<-operator

If the metacode has been enabled:

11. it has a virtual function _getdesc that returns the classes meta description
12. if it is a structured type, it has a virtual function _getref that returns a pointer to one of its components/members, specified through its name

If the Tcl code has been enabled:

13. it has a virtual function TclGetDesc to access the metacode's _getdesc routine from Tcl
14. it has a virtual function TclGetVal to retrieve an instance's value
15. it has a virtual function TclSetVal to change an instance's value
16. for SET, SEQUENCE, SET OF and SEQUENCE of: it has a virtual function TclUnsetVal to clear OPTIONAL members or to delete list elements, respectively

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

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

public:
                                    Foo();
                                    Foo (const Foo &);
                                     Foo();
    AsnType                         *Clone() const;
    Foo                             &operator = (const Foo &);

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

    // PDU (tags/lengths/content) encode and decode routines
    AsnLen                          BEnc (BUF_TYPE b);
    void                            BDec (BUF_TYPE b, 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) const;

#if META
    const AsnTypeDesc               *_getdesc() const;
    AsnType                         *_getref (const char *membername, bool create = false);
#if TCL
    int                             TclGetDesc (Tcl_DString *) const;
    int                             TclGetVal (Tcl_Interp *) const;
    int                             TclSetVal (Tcl_Interp *, const char *valstr);
    int                             TclUnsetVal (Tcl_Interp *, const char *membername);
#endif
#endif
};
```

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 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, a 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 7.4 and 7.14. 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 a 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.

The meta routines and the Tcl interface will be described in chapters 8 and 9, respectively.

### 6.3.1 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();
                    T1 (const T1 &);
                     T1();
```

```
    AsnType                     *Clone() const;

    T1                          &operator = (const T1 &);

    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) const;

#if META
    static const AsnSequenceTypeDesc
                                _desc;
    static const AsnSequenceMemberDesc
                                mdescs[];
    const AsnTypeDesc           *_getdesc() const;
    AsnType                     *_getref (const char *membername, bool create = false);

#if TCL
    int                         TclGetDesc (Tcl_DString *) const;
    int                         TclGetVal (Tcl_Interp *) const;
    int                         TclSetVal (Tcl_Interp *, const char *valstr);
    int                         TclUnsetVal (Tcl_Interp *, const char *membname);
#endif // TCL
#endif // META
};
```

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.

### 6.3.2 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 to 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
    {
        AsnInt              *integer;
        AsnOid              *oid;
    };

                            T1Choice();
                            T1Choice (const T1Choice &);
                             T1Choice();
    AsnType                 *Clone() const;

    T1Choice                &operator = (const T1Choice &);

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

    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) const;

#if META
    static const AsnChoiceTypeDesc  _desc;
    static const AsnChoiceMemberDesc
                            _mdescs[];

    const AsnTypeDesc       *_getdesc() const;
    AsnType                 *_getref (const char *membername, bool create = false);

#if TCL
```

```
    int                         TclGetDesc (Tcl_DString *) const;
    int                         TclGetVal (Tcl_Interp *) const;
    int                         TclSetVal (Tcl_Interp *, const char *valstr);
#endif // TCL
#endif // META
};
```

### 6.3.3 SET OF and SEQUENCE OF

Each SET OF and SEQUENCE OF type produces its own list class, unlike the C back-end 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() { count = 0; first = curr = last = NULL; }
                                 T1SeqOf();
    AsnType                     *Clone() const;

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

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

    // 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

    // removing the current elmt from the list
    void                            RemoveCurrFromList();

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

    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);

#if META
    static const AsnListTypeDesc    _desc;
    const AsnTypeDesc               *_getdesc() const;
    AsnType                         *_getref (const char *index, bool create = false);

#if TCL
    int                             TclGetDesc (Tcl_DString *) const;
    int                             TclGetVal (Tcl_Interp *) const;
    int                             TclSetVal (Tcl_Interp *, const char *valstr);
    int                             TclUnsetVal (Tcl_Interp *, const char *valstr);
#endif // TCL
#endif // META
};
```

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 to the current element of the list. Several routines are provided 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 to 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 to 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 to 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() const;
```

This returns the number of elements in the list.

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

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 the 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 the list and returns a pointer to the new 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 the new 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 the new 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.

### 6.3.4 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 a 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:
#if TCL
                                        T1Enum(): AsnEnum (_nmdescs[0].value) {}
#else
                                        T1Enum(): AsnEnum () {}
#endif
                                        T1Enum (int i): AsnEnum (i) {}
      enum
      {
            a = 0,
            b = 1,
            c = 2
      };

#if META
      static const AsnNameDesc          _nmdescs[];
      static const AsnEnumTypeDesc      _desc;
      const AsnTypeDesc                 *_getdesc() const;
#endif // META
};
```

## 6.4 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;
```

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 (true);
```

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.

## 6.5 Compiler Directives

Compiler directives are ignored by the C++ backend of snacc. If you want to implement them, look at the FillCxxTypeDefInfo routine in file `.../compiler/back-ends/c++-gen/types.c`. Then look at the way it is done for the C back-end (file `.../compiler/back-ends/c-gen/type-info.c`)

## 6.6 Compiling the Generated C++ Code

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

1. The include directory where the files from `.../c++-lib/inc/` have been installed in your include path so that the C++ sources can include these library header files. The header files should be included with statements like #include <snacc/c++/asn-incl.h> and your C++ compiler should be supplied with `-I/usr/local/include` in case snacc got installed under `/usr/local/`.
2. to link with the C++ ASN.1 runtime library, `.../c++-lib/libasn1c++.a`. In case snacc got installed under `/usr/local/`, your linker may need to be supplied with `-L/usr/local/lib` and `-lasn1c++` as arguments.
3. to link with the math library (`-lm`), since the ASN.1 REAL type's encode and decode routine use some math routines.

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

# Chapter 7

# C++ ASN.1 Library

## 7.1 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 `.../c++-lib/inc/` and `.../c++-lib/src/`.

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 6.3.

The top level PDU encode and decode methods are the same for all library types so they are defined as macros in `.../c++-lib/inc/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 `.../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.

## 7.2 Tags

The C++ tags are identical to those used in snacc's C ASN.1 environment. While it would have been nice to 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 5.2.

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.

## 7.3 Lengths

The C++ representation of lengths is the same at the C representation described in Section 5.3. The length type was not given its own C++ class for reasons similar to those of tags.

## 7.4 The AsnType Base Class

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

- the destructor
- Clone()
- BDec()
- BEnc()
- Print()
- _getdesc() (metacode)
- _getref() (metacode)
- TclGetDesc() (Tcl interface)
- TclGetVal() (Tcl interface)
- TclSetVal() (Tcl interface)
- TclUnsetVal() (Tcl interface)

The AsnType class is defined as follows:

```
class AsnType
{
public:
    virtual                         AsnType();

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

#if META
    static const AsnTypeDesc        _desc;

    virtual const AsnTypeDesc       *_getdesc() const;
    virtual AsnType                 *_getref (const char *membername, bool create=false);

private:
    const char                      *_typename() const;

#if TCL
public:
    virtual int                     TclGetDesc (Tcl_DString *) const;
    virtual int                     TclGetVal (Tcl_Interp *) const;
    virtual int                     TclSetVal (Tcl_Interp *, const char *val);
    virtual int                     TclUnsetVal (Tcl_Interp *, const char *membernames);
#endif // TCL
#endif // META
};
```

The AsnType class and its virtual functions were added to support the ANY DEFINED BY type handling mechanism. This mechanism is described in Section 7.14.

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. Unfortunately, calls to virtual functions are slower than calls to normal functions due to 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 pages 215–217 of Stroustrup [15] for a discussion of this.

## 7.5 BOOLEAN

The BOOLEAN type is represented by the AsnBool class. The following is the class definition of AsnBool from the `.../c++-lib/inc/asn-bool.h` file.

```
class AsnBool: public AsnType
{
protected:
    bool                        value;

public:
                                AsnBool (const bool val): value (val) {};
                                AsnBool() {};
    AsnType                     *Clone() const;
                                operator bool() const { return value; }
    AsnBool                     &operator = (bool newvalue) { value = newvalue; return *this; }

    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) const;

#if META
    static const AsnBoolTypeDesc _desc;

    const AsnTypeDesc           *_getdesc() const;

#if TCL
    int                         TclGetVal (Tcl_Interp *) const;
    int                         TclSetVal (Tcl_Interp *, const char *val);
#endif // TCL
#endif // META
};
```

The upcoming C++ standard [5] defines a type bool accompanied by false and true to denote the boolean values. The Snacc configuration script checks whether the C++ compiler already supplies this new type and defines a look-alike if it does not.

For backwards compatibility, AsnBool::true and AsnBool::false are still valid.

The operator bool() is defined such that when an AsnBool value is cast to a boolean, it returns the C++ style boolean value of the AsnBools value. There is also a constructor for AsnBool that builds an AsnBool value from the given C++ style 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;
    bool                    connectionOpen;
    bool                    pduOk;
    ...
    okToSend = connectionOpen && pduOk; // assign AsnBool from bool
    if (okToSend) // cast AsnBool to bool
        ...
}
```

The AsnBool class contains the standard encode and decode methods that were described in Chapter 6.

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.

## 7.6 INTEGER

The INTEGER type is represented by the AsnInt class. The following is the class definition of AsnInt from the `.../c++-lib/inc/asn-int.h` file.

```
class AsnInt: public AsnType
{
protected:
    AsnIntType              value;

public:
                            AsnInt() {}
                            AsnInt (AsnIntType val): value (val) {}

    AsnType                 *Clone() const;

                            operator AsnIntType() { return value; }
    AsnInt                  &operator = (AsnIntType newvalue) { value = newvalue; return *this; }
    void                    Set (AsnIntType i) { value = i; }
    void                    ReSet (AsnIntType i) { value = i; }

    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) const { os << value; }

#if META
    static const AsnIntTypeDesc _desc;

    const AsnTypeDesc           *_getdesc() const;

#if TCL
    int                         TclGetVal (Tcl_Interp *) const;
    int                         TclSetVal (Tcl_Interp *, const char *val);
#endif /* TCL */
#endif /* META */
};
```

The internal representation of an ASN.1 INTEGER value is a AsnIntType. This is a typedef, the real type may be int, long or short, whatever is found to be 32 bits in size. The types' sizes depend on the machine and compiler and are determined at configuration time. The BDecContent routine will signal an error if the integer value being decoded will not fit into the AsnIntType representation.

Unlike the C ASN.1 library, the non-negative 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 if your ASN.1 source uses subtyping similar to:

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

## 7.7 ENUMERATED

The ENUMERATED type is represented by the AsnEnum class. The following is the class definition of AsnEnum from the `.../c++-lib/inc/asn-enum.h` file.

```
class AsnEnum: public AsnInt
{
public:
#if !TCL
                                AsnEnum(): AsnInt() {}
#endif
                                AsnEnum (int i): AsnInt (i) {}

    AsnType                     *Clone() const;

    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);

#if META
    static const AsnEnumTypeDesc _desc;

    const AsnTypeDesc           *_getdesc() const;
#endif /* META */
};
```

Note that it is not derived from class AsnType directly but from class AsnInt instead.

## 7.8 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() {}
    AsnType                     *Clone() const;

    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) const { os << "NULL"; }

#if META
    static const AsnNullTypeDesc _desc;

    const AsnTypeDesc           *_getdesc() const;

#if TCL
    int                         TclGetVal (Tcl_Interp *) const;
    int                         TclSetVal (Tcl_Interp *, const char *val);
#endif /* TCL */
#endif /* META */
};
```

## 7.9 REAL

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

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

public:
                                    AsnReal(): value (0.0) {}
                                    AsnReal (double val): value (val) {}
    AsnType                         *Clone() const;
                                    operator double() const { return value; }
    AsnReal                         &operator = (double newvalue) { value = newvalue; return *this; }

    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) const { os << value; }

#if META
    static const AsnRealTypeDesc    _desc;

    const AsnTypeDesc               *_getdesc() const;

#if TCL
    int                             TclGetVal (Tcl_Interp *) const;
    int                             TclSetVal (Tcl_Interp *, const char *val);
#endif /* TCL */
#endif /* META */
};

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 de-

fault encode routine uses frexp. Currently, the `.../configure` script has not got any checks for the IEEE format or library and therefore does not define any of the symbols. (This should be fixed.)

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 of a REAL, not the decimal encoding.

## 7.10 BIT STRING

The BIT STRING type is represented by the AsnBits class. From `.../c++-lib/inc/asn-bits.h`:

```
class AsnBits: public AsnType
{
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);
protected:
    size_t                      bitLen;
    char                        *bits;

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

    AsnType                     *Clone() const;

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

    size_t                      BitLen() { return bitLen; }

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

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

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

    void                            SetBit (size_t);
    void                            ClrBit (size_t);
    int                             GetBit (size_t) const;

    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) const;

#if META
    static const AsnBitsTypeDesc    _desc;

    const AsnTypeDesc               *_getdesc() const;

#if TCL
    int                             TclGetVal (Tcl_Interp *) const;
    int                             TclSetVal (Tcl_Interp *, const char *val);
#endif /* TCL */
#endif /* META */
};
```

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 be created three ways: from the number of bits, 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 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 5.8). Snacc provides no facility for encoding or internally representing constructed BIT STRING values.

## 7.11 OCTET STRING

OCTET STRING values are represented with the AsnOcts class. From `.../c++-lib/inc/asn-octs.h`:

```
class AsnOcts: public AsnType
{
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);

protected:
    size_t                      octetLen;
    char                        *octs;

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 size_t len) { Set (str, len); }
                                AsnOcts (const AsnOcts &o) { Set (o); }
                                 AsnOcts() { delete octs; }
    AsnType                     *Clone() const;

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

    size_t                      Len() const { return octetLen; }
                                operator const char *() const { return octs; }
                                operator char *() { return octs; }

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

    // these set the octs and octetLen values
    void                        Set (const char *str, size_t 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, size_t len);
    void                        ReSet (const AsnOcts &o);
    void                        ReSet (const char *str);
```

```
    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) const;

#if META
    static const AsnOctsTypeDesc  _desc;

    const AsnTypeDesc           *_getdesc() const;

#if TCL
    int                         TclGetVal (Tcl_Interp *) const;
    int                         TclSetVal (Tcl_Interp *, const char *val);
#endif /* TCL */
#endif /* META */
};
```

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 NUL 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 string (this does not usually include the NUL 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 NUL 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. The Len method returns the length in bytes of the string value. These may be useful for passing the octets to other functions such as memcpy etc.

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

## 7.12 OBJECT IDENTIFIER

OBJECT IDENTIFIER values are represented with the AsnOid class. From `.../c++-lib/inc/asn-oid.h`:

```
class AsnOid: public AsnType
{
private:
    int                         OidEquiv (AsnOid o);

protected:
    size_t                      octetLen;
    char                        *oid;

public:
                                AsnOid(): oid (NULL), octetLen (0) {}
                                AsnOid (const char *encOid, size_t 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; }
    AsnType                     *Clone() const;

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

    size_t                      Len() { return octetLen; }
    const char                  *Str() const { return oid; }
                                operator const char * () const { return oid; }
                                operator char * () { return oid; }
    unsigned long int           NumArcs() const;

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

    // Set methods overwrite oid and octetLen values
    void                        Set (const char *encOid, const size_t 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 size_t 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);

    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) const;

#if META
    static const AsnOidTypeDesc _desc;

    const AsnTypeDesc           *_getdesc() const;

#if TCL
    int                         TclGetVal (Tcl_Interp *) const;
    int                         TclSetVal (Tcl_Interp *, const char *val);
#endif /* TCL */
#endif /* META */
};
```

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 than the others.

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

## 7.13 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 was described in the previous chapter.

## 7.14 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 `.../c++-lib/inc/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) const { 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 5.12 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:

- the destructor
- Clone()
- BDec()
- BEnc()
- Print()
- _getdesc() (metacode)
- _getref() (metacode)
- TclGetDesc() (Tcl interface)
- TclGetVal() (Tcl interface)
- TclSetVal() (Tcl interface)
- TclUnsetVal() (Tcl interface)

This allows the ANY DEFINED BY handling routines to treat a value of any ASN.1 type as an AsnType. So, for each type the ANY DEFINED BY handling code has access to the 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 no 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.

## 7.15 Buffer Management

The C++ buffer management provided with snacc is similar to that of the C SBufs. The following is from `.../c++-lib/inc/asn-buf.h`:

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

```
    bool                        readError;

public:
    void                        Init (char *data, size_t dataLen);
    void                        ResetInReadMode();
    void                        ResetInWriteRvsMode();
    void                        InstallData (char *data, size_t dataLen);
    size_t                      DataLen();
    char                        *DataPtr();
    size_t                      BlkLen();
    char                        *BlkPtr();
    bool                        Eod();

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

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

## 7.16 Dynamic Memory Management

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

To change new and delete to use your own memory management scheme 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 [15].

## 7.17 Error Management

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

# Chapter 8

# The Metacode

## 8.1 Introduction

When you call snacc, during its compilation, the text in the `.asn1` files gets turned into e.g. C++ classes. Names become identifiers, and after the C++ compilation, the user program has no more access to the original module and type names, only to pointers and the bits and bytes of the classes' contents.

The metacode remedies this. Using it, a program can access the modules, their types, their subtypes and the named values via strings. Generic programs do not have to know any of the modules' or types' names—all the information can be traversed starting at a single well-known place.

The metacode has to map strings ("component "name" in type "Person" in ASN.1 module "Mail"") into the in-core address of the indicated object. Moreover, using the metacode, a generic program has to be able to traverse all modules, types and their components to learn about their names and types. Given the name of a type, the metacode must be able to return a newly allocated object instance.

The metacode is an extension to the Snacc compiler's C++ backend and the C++ runtime library.

Since the metacode relies heavyly on the virtual function call mechanism, it is only implemented for the C++ backend.

A number of functions has been added to the C++ runtime library. All code extensions have been wrapped into preprocessor conditionals. Currently, only one or two libraries are made in `.../c++-lib/`, one with neither metacode nor Tcl interface, and an optional additional one with both extensions. If you cannot or do not want to (as stated in `.../policy.h`) use the metacode or the Tcl interface, Snacc will be compiled without it, and only the normal library will get made.

If you want to use the metacode but not the Tcl interface, you have got to change the makefile to compile another set of `.o` files in a directory you may want to name `.../c++-lib/meta/` and archive the resulting files in `libasn1meta.a`. Com-

pile with `-DMETA=1 -DTCL=0`.

## 8.2 Implementation

With the metacode, the strings and the ASN.1 type components' type information are stored in arrays, one per structured ASN.1 type. The type descriptions are listed in another array, one per ASN.1 module.

In the C++ code generated by snacc, every ASN.1 type is represented by a C++ class. These C++ classes for ASN.1 simple types are implemented in the runtime library, and for structured types they are generated by the backend. The metacode is an extension to all of those C++ classes. The metacode put into every C++ class is very similar:

- a static _desc member is always present.
- simple types with names (ENUMERATED, INTEGER, BIT STRING) get an additional static _nmdescs[] member. The array is exclusively referenced from _desc. The array provides the bidirectional mapping of symbolic and numeric values.
- structured types with members (SET, SEQUENCE, CHOICE) get an additional static _mdescs[] member. The array is exclusively referenced from _desc. The array references the components' type descriptions (their _desc data members).
- every class gets a virtual _getref() function. Its only purpose is to return the address of _desc[1].
- structured types with members (SET, SEQUENCE, CHOICE) get an additional virtual _getref() function. This function provides the member name to member address mapping.

The data members are static, and therefore get instantiated exactly once per executable, not once per class object instance.

To get an impression, let us have a look at an example: The two ASN.1 types

```
File ::= SET
{
  name [0] PrintableString,
  contents [1] OCTET STRING,
  checksum [2] INTEGER OPTIONAL,
  read-only [3] BOOLEAN DEFAULT FALSE
}

Directory ::= SET
{
  name PrintableString,
  files SET OF File
}
```

[1] This sounds as if virtual data members were a nice idea, and in fact they are. The C++ standards committees are currently discussing this.

get turned into two individual C++ classes:

```
class File: public AsnType
{
public:
        PrintableString                 name;
        AsnOcts                         contents;
        AsnInt                          *checksum;
        AsnBool                         *read_only;

#if META
        static const AsnSetTypeDesc     _desc;
        static const AsnSetMemberDesc   _mdescs[];
        const AsnTypeDesc               *_getdesc() const;
        AsnType                         *_getref (const char *membername, bool create = false);
#endif // META

        // ...other functions omitted...
};

class Directory: public AsnType
{
public:
        PrintableString                 name;
        DirectorySetOf                  files;

#if META
        static const AsnSetTypeDesc     _desc;
        static const AsnSetMemberDesc   _mdescs[];
        const AsnTypeDesc               *_getdesc() const;
        AsnType                         *_getref (const char *membername, bool create = false);
#endif // META

        // ...other functions omitted...
};
```

The above definitions stem from the `.h` file, the following code is taken from the `.C` file. Only the code for the Directory type is shown, because the code for the File type looks very similar.

```
#if META

static AsnType *createDirectory()
{
        return new Directory;
}

const AsnSetMemberDesc Directory::_mdescs[] =
{
        AsnSetMemberDesc ("name", &PrintableString::_desc, false), // `name'
        AsnSetMemberDesc ("files", &DirectorySetOf::_desc, false), // `files'
        AsnSetMemberDesc()
```

```
};

const AsnSetTypeDesc Directory::_desc
(
        &EdEx_StructuredModuleDesc,
        "Directory", // `Directory'
        true,
        AsnTypeDesc::SET,
        createDirectory,
        _mdescs
);

const AsnTypeDesc *Directory::_getdesc() const
{
        return &_desc;
}

AsnType *Directory::_getref (const char *membername, bool create)
{
        if (!strcmp (membername, "name"))
                return &name;
        if (!strcmp (membername, "files"))
                return &files;
        return NULL;
}

#endif // META
```

The two ASN.1 types get turned into two individual C++ classes, but their _desc members point to two different instances of the same type. The C++ backend generates an individual C++ class for every structured ASN.1 type. The metacode is different: The types of the descriptions data members (_desc, _nmdesc and _mdesc) despend on the *general* ASN.1 type (i.e. e.g. SET, not File).

Every `.asn1` file contains an ASN.1 module and gets translated into a `.C` file that contains an array that lists all the module's type descriptions.

Snacc generates an additional file `modules.C` that contains an array that lists all the module descriptions. This single top level array provides the well known entry point for the whole module and type hierarchy.

The type of the _desc member differs depending on the ASN.1 type it describes. The different _desc types mirror the AsnType class hierarchy. For example, the ASN.1 BOOLEAN type is mapped into a C++ class called AsnBool and is described by a C++ class AsnBoolTypeDesc. The root of the _desc class hierarchy is called AsnTypeDesc and looks as follows (taken from `.../c++-lib/inc/meta.h`):

```
struct AsnTypeDesc
{
        const AsnModuleDesc             *module;
        const char                      *const name; // NULL for basic types
        const bool                      pdu;
        const enum Type         // NOTE: keep this enum in sync with the typenames[]
```

```
    {
        VOID,
        ALIAS,

        INTEGER,
        REAL,
        NUL_, // sic! (can't fight the ubiquitous NULL #define)
        BOOLEAN,
        ENUMERATED,
        BIT_STRING,
        OCTET_STRING,
        OBJECT_IDENTIFIER,

        SET,
        SEQUENCE,
        SET_OF,
        SEQUENCE_OF,
        CHOICE,
        ANY,
    }                               type;

    static const char               *const typenames[];

                                    AsnTypeDesc (const AsnModuleDesc *, const char *,
                                                 bool ispdu, AsnType *(*create)(), Type);

    AsnType                         *(*create)();

    virtual const AsnModuleDesc     *getmodule() const;
    virtual const char              *getname() const;
    virtual bool                    ispdu() const;
    virtual Type                    gettype() const;
    virtual const AsnNameDesc       *getnames() const;

#if TCL
    virtual int                     TclGetDesc (Tcl_DString *) const;
    virtual int                     TclGetDesc2 (Tcl_DString *) const;
#endif
};

typedef AsnTypeDesc                 AsnRealTypeDesc;
typedef AsnTypeDesc                 AsnNullTypeDesc;
typedef AsnTypeDesc                 AsnBoolTypeDesc;

typedef AsnTypeDesc                 AsnOctsTypeDesc;
typedef AsnTypeDesc                 AsnOidTypeDesc;
```

AsnTypeDesc's data members provide the following information:

The module data member points to the module description described at the end of this chapter in section 8.2.6.

The name is either the type's name as used by the backend code (default, or if snacc was called with the `-mC` switch) or the type's given names as defined in the

`.asn1` file (if snacc has been called with the `-mA` command line option). Section 6.2 on page 77 explains the differences. (For example, the backend code sometimes has an additional number tacked to the name—you can see the effect in figure 10.1 on page 131.) The generated source code contains the respective counterpart printed in a comment.

The pdu flag is set to true iff the type was listed after snacc's `-meta` or `-tcl` switch.

The type member is used as an index into the typenames array—the virtual function call mechanism obliviates the use for any switch statements.

The create data member points to a global function that returns a pointer to a newly allocated object of the description type's mirror type, that is, gives you an instance for the generic description. It is the counterpart to the AsnType's _getdesc function which goes in the opposite direction, from the object instance to its generic description. The AsnType's Clone function serves a similar purpose as the AsnTypeDesc's create function.

The AsnTypeDesc class is the only class in the hierarchy that has got the module, name, pdu and type data members, and AsnNamesTypeDesc the only class to implement a names data member. Therefore, unlike _getdesc() mentioned above, the five virtual functions getmodule, getname, ispdu, gettype and getnames are not meant to implement some kind of virtual data members, but help to implement the alias type description functionality described in section 8.2.4.

As you can see looking at the last five code lines with the typedefs, the five ASN.1 simple types REAL, NULL, BOOLEAN, OCTET STRING and OBJECT IDENTIFIER are directly described by instances of this class. The other types, having either named values or components, are more demanding and have their own classes derived from AsnTypeDesc.

### 8.2.1 Named Values

Some basic ASN.1 types allow values to be named, namely INTEGER, ENUMERATED and BIT STRING. The accompanying description types contain arrays listing the names and values. The virtual function getnames() returns this array. The respective C++ classes in the runtime library contain an AsnNameDesc _nmdescs[] array, the address of which is given to the type descriptions constructor as last argument.

```
struct AsnNameDesc
{
    const char                  *const name;
    const long int              value;
};

struct AsnNamesTypeDesc: AsnTypeDesc
{
    const AsnNameDesc           *const names;

                                AsnNamesTypeDesc (const AsnModuleDesc *, const char *,
```

```
                                        bool ispdu, AsnType *(*create)(), Type, const AsnNameDesc *);

    const AsnNameDesc               *getnames() const;

#if TCL
    int                             TclGetDesc (Tcl_DString *) const;
    // for BIT STRING and INTEGER, ENUMERATED has its own:
    int                             TclGetDesc2 (Tcl_DString *) const;
#endif
};

struct AsnEnumTypeDesc: AsnNamesTypeDesc
{
                                    AsnEnumTypeDesc (const AsnModuleDesc *, const char *,
                                      bool ispdu, Type, AsnType *(*create)(), const AsnNameDesc *);

#if TCL
    int                             TclGetDesc2 (Tcl_DString *) const;
#endif
};

typedef AsnNamesTypeDesc            AsnIntTypeDesc;
typedef AsnNamesTypeDesc            AsnBitsTypeDesc;
```

The ENUMERATED type gets its own description class because the Tcl interface for ENUMERATED types behaves differently than for the INTEGER and BIT STRING types.

As for AsnTypeDesc::name above, the content of AsnNameDesc::name is either the value's name as used by the backend code (default, or if snacc was called with the `-mC` switch) or the value's name as given in the `.asn1` file (if snacc has been called with the `-mA` command line option). The generated source code contains the respective counterpart printed in a comment.

### 8.2.2 Types with Members

The ASN.1 types CHOICE, SET and SEQUENCE are defined in terms of other types, their so-called components. The ASN.1 components map into C++ data members.

The three ASN.1 structured types get mapped into C++ classes that contain an Asn…MemberDesc _mdescs[] array (with the `…' replaced by `Choice' `Set' or `Sequence'). The address of this array is given to the description type's constructor as last argument. The elements of this array point to the descriptions of the data classes data members. This is similar to the named values above, only the integral value has been replaced by a pointer to a type description.

```
struct AsnMemberDesc // description of CHOICE member; base class for AsnSe_MemberDesc
{
    const char                      *const name;
    const AsnTypeDesc               *const desc;
```

```
                                        AsnMemberDesc (const char *, const AsnTypeDesc *);
                                        AsnMemberDesc();

#if TCL
        virtual int                     TclGetDesc (Tcl_DString *) const;
        virtual int                     TclGetDesc2 (Tcl_DString *) const;
#endif
};

struct AsnSe_MemberDesc: AsnMemberDesc      // _ == t/quence; description of SET or SEQUENCE member
{
        bool                            optional;

                                        AsnSe_MemberDesc (const char *, const AsnTypeDesc *, bool);
                                        AsnSe_MemberDesc();

#if TCL
        int                             TclGetDesc2 (Tcl_DString *) const;
#endif
};

typedef AsnMemberDesc                   AsnChoiceMemberDesc;
typedef AsnSe_MemberDesc                AsnSetMemberDesc;
typedef AsnSe_MemberDesc                AsnSequenceMemberDesc;

struct AsnMembersTypeDesc: AsnTypeDesc
{
                                        AsnMembersTypeDesc (const AsnModuleDesc *, const char *,
                                                            bool ispdu, AsnType *(*create)(), Type);

#if TCL
        int                             TclGetDesc (Tcl_DString *) const;
#endif
};

struct AsnChoiceTypeDesc: AsnMembersTypeDesc
{
        const AsnChoiceMemberDesc       *const members;

                                        AsnChoiceTypeDesc (const AsnModuleDesc *, const char *,
                                    bool ispdu, AsnType *(*create)(), Type, const AsnChoiceMemberDesc *);

        int                             choicebyname (const char *name) const;
        const char                      *choicebyvalue (int value) const;

#if TCL
        int                             TclGetDesc2 (Tcl_DString *) const;
#endif
};

struct AsnSe_TypeDesc: AsnMembersTypeDesc       // _ == t/quence
{
        const AsnSe_MemberDesc          *const members;
```

```
                                    AsnSe_TypeDesc (const AsnModuleDesc *, const char *,
                                bool ispdu, AsnType *(*create)(), Type, const AsnSe_MemberDesc *);

#if TCL
    int                             TclGetDesc2 (Tcl_DString *) const;
#endif
};

typedef AsnSe_TypeDesc              AsnSetTypeDesc;
typedef AsnSe_TypeDesc              AsnSequenceTypeDesc;
```

As for AsnTypeDesc::name above, the content of AsnMemberDesc::name is either the member's name as used by the backend code (default, or if snacc was called with the `-mC` switch) or the component's name as defined in the `.asn1` file (if snacc has been called with the `-mA` command line option). The generated source code contains the respective counterpart printed in a comment. In case the ASN.1 component was not given a name, the backend's member name is used instead.

The data classes have a member function called _getref, that allows the C++ class members to be accessed by their name. _getref() is the second metacode function and it is present in all C++ classes representing composed ASN.1 types.

A class for a SET contains the following code fragment:

```
class FooSet: public AsnType
    AsnInt                          bar; // an example data member
    ...// lots of member functions
#if META
    static const AsnSetTypeDesc     _desc;
    static const AsnSetMemberDesc   mdescs[];
    const AsnTypeDesc               *_getdesc() const;
    AsnType                         *_getref (const char *membername, bool create = false);
#if TCL
    int                             TclGetDesc (Tcl_DString *) const;
    int                             TclGetVal (Tcl_Interp *) const;
    int                             TclSetVal (Tcl_Interp *, const char *valstr);
    int                             TclUnsetVal (Tcl_Interp *, const char *membername);
#endif // TCL
#endif // META
};
```

_getref()'s bool parameter create determines whether a non-existing member should be returned as a NULL pointer or whether it should instead be allocated and its address be returned. This parameter is used by value reading and writing routines to implement their different member access semantics.

The following four assignments are equivalent:

```
FooSet foo;
foo.bar = 1;
(AsnInt *)foo._getref ("bar") = 1;
foo.bar.TclSetVal (interp, "1");
foo._getref ("bar")->TclSetVal (interp, "1");
```

TclSetVal() is a virtual member function and therefore no cast from AsnType * to AsnInt * is required. The Tcl interface will be described in chapter 9.

The C++ classes that represent CHOICE types contain an enum ChoiceIdEnum that allows _getref() to be written using a switch statement. The functions choicebyname() and choicebyvalue() turn the component's name into its enumeration value and vice versa. (The enum has not been introduced with the metacode, it is used by Snacc's encoding and printing functions as well.)

### 8.2.3 SET OF and SEQUENCE OF

The list description behaves like an ASN.1 simple type's—the description type is derived directly from the type descriptions' base class and does not redefine any of the metacode functions:

```
struct AsnListTypeDesc: AsnTypeDesc
{
    const AsnTypeDesc           *const base;

                                AsnListTypeDesc (const AsnModuleDesc *, const char *,
                                    bool ispdu, Type, AsnType *(*create)(), const AsnTypeDesc *);

#if TCL
    int                         TclGetDesc (Tcl_DString *) const;
#endif
};
```

The TclGetDesc function merely adds the base type's standard type description (module and type name, pdu flag and type) after its own, so that a programmer may take the base type's name and ask the metacode once again for the base type's full description.

A list type's data class on the other hand has got a _getref() function that gives access to the list's elements and it can be used to insert new elements at any desired position.

### 8.2.4 Aliases

For ASN.1 types being defined as a direct copy of another type, snacc in normal operation uses a C++ typedef to define the C++ type. Since this typedef makes the two types totally equivalent, the metacode has no chance to preserve the two types' different names and thus, this contruct cannot be used. A new C++ class has got to be defined instead.

Example: the following ASN.1 code snippet...

```
Int1 ::= INTEGER { foo(42) }
Int2 ::= Int1
```

...maps into the following C++ definitions:

```
class Int1: public AsnInt
{
public:
                                        Int1(): AsnInt() {}
                                        Int1 (int i): AsnInt (i) {}
        enum
        {
                foo = 42
        };

#if META
        static const AsnNameDesc        _nmdescs[];
        static const AsnIntTypeDesc     _desc;
        const AsnTypeDesc               *_getdesc() const;
#endif // META
};

#if META
struct Int2: public Int1
{
                                        Int2(): Int1() {}
                                        Int2 (int i): Int1 (i) {}
        AsnType                         *Clone() const;

        static const AsnAliasTypeDesc   _desc;
        const AsnTypeDesc               *_getdesc() const;
};

#else // META

typedef Int1                            Int2;

#endif // META
```

The descriptor type's definition points to the reference type:

```
struct AsnAliasTypeDesc: AsnTypeDesc
{
        const AsnTypeDesc               *const alias;

                                        AsnAliasTypeDesc (const AsnModuleDesc *, const char *,
                                            bool ispdu, AsnType *(*create)(), Type, const AsnTypeDesc *);

        const AsnModuleDesc             *getmodule() const;
        const char                      *getname() const;
        bool                            ispdu() const;
        Type                            gettype() const;
        const AsnNameDesc               *getnames() const;

#if TCL
        int                             TclGetDesc (Tcl_DString *) const;
#endif
};
```

The AsnAliasTypeDesc is the reason for the five virtual functions from getmodule to getnames defined in both AsnTypeDesc and AsnNamesTypeDesc on the one hand and AsnAliasTypeDesc on the other hand. While the alias type belongs to a different module or has another type name, and it may have another pdu flag value, its type and names array values are those of its reference type. Therefore, AsnAliasTypeDesc's first three functions of return the description's own values, and the latter two call their reference type's functions.

The getnames function has to be defined in the hierarchy's base class because the aliases may be defined for any type of type, not only for types with named values.

### 8.2.5 ANY (DEFINED BY)

ANY DEFINED BY is quite problematic. The ASN.1 Book [14] calls it "a rather half-baked attempt at solution". Since snacc has problems with it—the user has to modify the snacc generated code—and none of our applications requires this construct, no effort has been made to implement it.

ANY itself on the other hand would be quite simple to implement—the virtual function call mechanism that is used to implement the ANY type is the basis for the metacode as well. But again, since we have no need for the ANY type, it is as far unimplemented. Besides that, according to the ASN.1 book, the "use of ANY without the DEFINED BY construct is "deprecated" (frowned upon) by the standard". The next ASN.1 standard will probably not have the ANY type any more. In the 1993 draft standard [4], ANY and ANY DEFINED BY can be found in "Annex I: Superseded features", Section 3: "The any type".

### 8.2.6 Modules

Every `.C` file (that corresponds to an `.asn1` file, or, an ASN.1 module), gets an array that lists all the module's types. This array contains pointers to all the _desc members of all classes of a module.

```
struct AsnModuleDesc
{
    const char                  *const name;
    const AsnTypeDesc           **const types;
};

extern const AsnModuleDesc      *asnModuleDescs[];
```

The modules themselves are listed in yet another array, the declaration of which is shown in the preceeding line. This array has got its own source file named `modules.C`. This array allows all modules to be found, and every type that is defined for these modules.

## 8.3 Efficiency

The metacode is designed with efficiency in mind. The metacode is intended for interpreted interfaces and therefore does not need to be highly optimized. On the other hand, the same object code should be useable for normal (non-metacode) tasks without loss of performance.

### 8.3.1 Normal Operation

The metacode does not significantly affect the normal mode of operation. The static data members `_mdescs` and `_desc` do not increase the class instances' size. The virtual function tables, which have already been present (they are used for the ANY type), get a little longer, but since these tables exist only once for every class, this difference is neglible. The class instances reference their virtual function table with a pointer, and so the metacode does not introduce any change here. Except for alias types, the C++ classes generated are exactly the same. The metacode introduces a new class for alias types, but since no new data members are introduced their size stays the same; only the virtual function table pointer is different.

All normal member functions (constructor, destructor, assignment operator, encode, decode and print functions) are identical—with only one exception: if the metacode is compiled to be usable by the Tcl interface, the constructors initialize their mandatory members.

To sum it up, both code and data grow, but except for a longer loading time from disk and an increased probability for cache misses, the code will run as fast as it does without the metacode.

### 8.3.2 Metacode

The metacode routines are kept quite simple. Intended to be used in conjunction with a Tcl interface, speed was not the most important concern. Consequently, the code is optimized more towards memory usage than run time efficiency. As an example, name to member resolution uses a linear lookup strategy instead of more elaborated algorithms like binary search or hash tables. I think for data types that typically have up to a dozen components, more sophisticated algorithms would have been overkill.

A typical object file gets almost 20% larger due to the metacode (the Tcl interface adds another 25%).

## 8.4 Metacode Vs. Type Tables

Here's a list of both the type tables' (see chapter 12) and the metacode's (dis)advantages:

- source code language:

  - The type tables are implemented for C only.
  - The metacode works only for C++.

- speed:
  - Encoding and decoding using the type tables is said to be about 4 times slower than using the C routines.
  + The metacode does not (significantly) harm performance.

- code size:
  + The tables are a lot smaller than the compiled routines.
  - The metacode makes the compiled code even larger.

- value constants:
  - The type tables lack the values defined in the `.asn1` files.
  + The metacode interacts fine with these values.

- named values:
  - The type tables lack the named values defined ENUMERATED, INTEGER and BIT STRING types.
  + The metacode interacts fine with these names.

- compatibility to normal snacc code:
  - The C structures defined by mkchdr and used by the type table encoding and decoding routines and the C structures defined by snacc's C backend are quite different.
  ± Where the backend's structures generated for SEQUENCE contain madatory members by value, the type table's structures contain only pointer members!

## 8.5 Setup for the Metacode Generator

To compile Snacc with the metacode generator, the following condition must be met:

- either the configure script must be able to find `tclsh` and the Tcl/Tk libraries or you have to insert a #define META 1 into `.../policy.h`
- the NO_META preprocessor macro in `.../policy.h` must not be set

# Chapter 9

# Tcl Interface

## 9.1 Introduction

This chapter describes the Snacc's Tcl interface, or: the metacode's link to the outside world.

Tcl is a simple scripting language which the author, John K. Ousterhout, describes in his book titled “Tcl and the TK Toolkit” [9]. Tcl's purpose is to be embedded into other applications, to provide a user interface by extending the language. Tk, an implementation of the Motif look and feel, is the first and best known extension to Tcl and is described in the same book.

Tcl has got only one data type, the NUL terminated character string. Tcl supports other data types like integers and lists, but they are represented as strings. A function operating on an integer first converts the string into an integer, performs its operation, converts the resulting value back into another string and returns it to the Tcl interpreter. Since lists and even the Tcl procedures are kept as strings, Tcl is rather slow. Computations in Tcl should best be kept at a minimum, and all intensive work should be wrapped into C or C++ functions and be made available as Tcl commands.

Since procedures and bodies of loops are kept in string form and parsed for every invocation, comments should be put outside code that is executed *very* often.

From Tcl's point of view, Snacc's Tcl interface is nothing but yet another Tcl extension. The Snacc Tcl interface extends the Tcl language by only one command, `snacc`. The first argument to this command specifies the action to be taken. This method is very practical for combining Tcl extensions since it avoids collisions with new command names from other extensions. For example, the Tcl core defines an `open` command. Snacc's Tcl interface wants to offer one as well and has to choose another name. This could have been done by naming it `snacc_open`, but I think it is better to stick to Tcl's well established convention and so the Tcl interface's open command became `snacc open`. To simplify the wording, I will refer to the `snacc subcommands' simply as `commands'.

The usual (non-metacode) snacc generated functions operate on memory buffers containing BER encoded data; they convert them into hierarchical C++ data structures and vice versa.

The Tcl interface is designed to allow controlled fine grained access to this hierarchical C++ data structure, to read and modify its contents. While both the C++ code and the Tcl look very similar, for example…

```
// this is C++ code
x–>foo–>bar = 42;
```

… and…

```
# this is Tcl code
snacc set {x foo bar} 42
```

… the C++ code gets compiled and the identifiers get turned into pointers and numeric offsets, and the Tcl code gets interpreted and has to mimic the C++ compiler at run time. This is what the metacode from chapter 8 is for.

To enable snacc's Tcl code generator, you have to give an additional `-tcl` option, followed by the list of PDU types. The `-meta` option can (and should) be omitted.

## 9.2 The `snacc` Tcl command

This section explains the Tcl (sub)commands provided by the Snacc extension. The commands are grouped in three catagories, commands operating on files (both their external and internal representation), commands accessing the meta information and commands operating on the content itself.

The file commands check the return value from system calls and behave like for example the Tcl `open` command, that is, they set the `errorCode` variable to `POSIX` *`errno`*, e.g. `POSIX ENOENT {No such file or directory}`.

The code should be fairly robust, not just against user and programmer errors from `outside' (using the `snacc` Tcl command), but against errors from the `inside' as well such as illegal numeric values for enumeration types or illegal choice settings as well.

There are two types of errors:

1. programmer errors, where the program has no other choice as to print a regret to the user and exit

2. user errors, such as trying to write to a read-only file, where the program should tell the user about their mistake and let them try something else.

The Tcl interface code helps the programmer for the second type of error by setting Tcl's `errorCode` variable. The program can `catch` any error, and, based on the `errorCode`, choose to deal with the mistake or rethrow the error that it is not prepared to handle.

### 9.2.1 File commands

Most snacc Tcl commands operate on so-called files. A file is an internal data structure that

- references the C++ representation of an ASN.1 data structure as a pointer to AsnType
- may be associated with an external file in the file system

The commands operating on these files are as follows:

**`snacc create`** ***`type`*** The command creates a file consisting only of an instance of type *type*. *type* has to be denoted as one argument, a Tcl list with two elements, module and type. No external filename is associated with this file.

**`snacc open`** ***`type filename`*** **`?`*****`flags`*****`?`** **`?`*****`mode`*****`?`** Open a file and read and decode its contents. *type* has to be denoted as one argument, a list with two elements, module and type. The optional *flags* may consist of:

**`create`** If the file does not exist, create it. If this flag is not given and the file does not already exist, an error occurs.

**`truncate`** If the file exists, drop its contents.

***`access`*** which may be either `ro` or `rw`, denoting read only and read/write access. If no access mode is specified, the file will be opened read/write if it is writable, and read only otherwise.

If the file is created, its mode is set to *mode*, minus umask, of course. *mode* may be any value accepted by Tcl_GetInt(3) (the function accepts octal values). At last, if the file could be opened, its contents is read and BER decoded. As for `snacc create` above, a file handle is returned.

If the file cannot be opened, an error is returned identical to Tcl's `open` command.

More errors can be returned, as described under `snacc read` below.

**`snacc close`** ***`file`*** closes the file *file* and invalidates the file handle.

**`snacc read`** ***`file`*** **`?`*****`type filename`*****`?`** without the *filename*, rereads the file from its old place; otherwise opens *filename*, reads its contents into *file* and closes it. The file's contents gets BER decoded.

In case no *filename* has been given but the *file* is not associated with a filename, an error is returned and `errorCode` is set to `SNACC MUSTOPEN`.

If Snacc's decoding routines detect an error, a Tcl error is returned and `errorCode` is set to `SNACC DECODE` *`errval`* where *errval* is the value returned by setjmp() (see sections 7.17 and 5.15 on pages 106 and 75, respectively).

If the input file is too short, the buffer will signal a read error and a Tcl error will be returned, with `errorCode` set to `SNACC DECODE EOBUF`.

**snacc write *file* ?*filename*?** BER encodes the file, then writes the file to its old place in case no *filename* has been given, or opens *filename*, writes *file* into it and closes it.

In case no *filename* has been given but the *file* is not associated with a filename, an error is returned and `errorCode` is set to `SNACC MUSTOPEN`. If you try to write to a read-only file, an error is returned and `errorCode` is set to `SNACC WRITE READONLY`.

**snacc finfo *file*** returns a list with two elements, the file name associated with it (the empty string if no external file name is associated with it) and an identifier which may be

**bad** the file is not associated with an external file.

**rw** the external file has been opened read/write.

**ro** the external file has been opened read only.

Since Tcl cannot operate on binary strings (that is, strings containing NUL bytes), but ASN.1 octet strings may contain arbitrary binary data, the binary data has to be converted into a replacement notation that Tcl can work with and that can be converted back to binary without loss of information. The conversion I chose is fairly simple: NUL is converted into a backslash followed by a zero digit, and every backslash is doubled.

These conversions for the most part take place automatically. In fact, there is only one point where the binary representation is necessary, when you want to read or write data from or into a file on disk. Two functions have been written to offer this: the export function converts and writes an octet string to an external file, and the import function reads binary data from a file and converts it to the Tcl compatible representation. Unlike the functions described above, these two do not operate on ASN.1 files, that is, the contents is not BER decoded/encoded, but may be used for any file in the file system.

**snacc import *filename*** opens the file named, reads its contents, closes it, performs the above described conversion and returns the resulting Tcl string.

**snacc export *string filename*** converts the Tcl string into its binary counterpart, opens the file named, writes the binary buffer into it and closes it. The file is created and truncated as necessary. The command returns the empty string.

### 9.2.2 Generic Information Retrieval

The following functions return information about the modules and their types. (This information is independent of any file instance, it is the information from the type descriptions in the `.asn1` files.)

**snacc modules** returns a list of module identifiers.

**`snacc types ?`*`module`*`?`** if a *module* is specified, returns a list of all type names of that module. otherwise, a list of all types is returned as a list of pairs, where each pair consists of the module name and the type name.

**`snacc type`** ***`type`*** where *type* is a list with two elements, module and type. This command returns a list with the following four elements:

0. the content type as a list consisting of module name and type name
1. an identifier that is either `pdu` or `sub` depending on the list of PDUs that had been given after snacc's `-tcl` option.
2. the ASN.1 type (e.g. INTEGER or CHOICE)
3. a list of items that depends on the ASN.1 type:

   **INTEGER** a (possibly empty) list of pairs of name and value for each named value.

   **ENUMERATED** a (non-empty) list of names.

   **SET, SEQUENCE** and **CHOICE** a list of lists of four elements similar to that being described here. Element 0 is the subtypes name, then follow content type (a pair consisting of module name and type name), *pdu* vs. *sub* and finally the ASN.1 type. (The fourth element of the outer list is omitted for obvious reasons: it would explode the type's description.)

### 9.2.3 Operations on Content and Structure

Finally, the last last four functions operate on the file instances itself. All four commands get a *path* argument that is constructed as follows:

- Every *path* starts with a file handle as returned by `snacc create` or `snacc open`.
- All subsequent path elements, except for the last, must indicate elements of composed types. For CHOICE, SET and SEQUENCE, these are member names, for SET OF and SEQUENCE OF, these are numeric indices.
- The last path element may reference a simple type.
- For SET OF and SEQUENCE OF, instead of a numeric index, a pair consisting of the word `insert` followed by a numeric index may be specified. In this case, a new list element is inserted before that addressed by the index. The index must be in the range $0 \ldots n-1$ to address existing elements and it must be in the range $0 \ldots n$ for insertion, where in both cases $n$ is the number of elements in the list.
- For `snacc unset`, the path must point to an optional member of a SET or SEQUENCE or to an element of a SET OF or SEQUENCE OF.

The commands are:

**snacc info *path*** returns information about the value pointed to by *path*. The information returned is quite similar to that of `snacc type` above, with the following exceptions:

- element 0, the content type, contains empty names for types that have not been given a name (e.g. a SET member of type OCTET STRING Example: the contents member in type File in file `edex1.asn1` (page 174) `snacc info` returns `{{} {}} sub {OCTET STRING}`).
- the number of elements depends on the ASN.1 type:

  simple types (**NULL**, **BOOLEAN**, **INTEGER**, **ENUMERATED**, **REAL**, **BIT STRING** and **OCTET STRING**): no additional elements are returned. For the list of named values for INTEGER, ENUMERATED and BIT STRING, you have to call `snacc type [lindex [snacc info` *path*`] 0]`, unless the content type equals `{{} {}}`.

  **CHOICE** a total of five elements is returned, number 3 is the name of the choice member currently chosen, and the final element number 4 is an identifier that is either `void` or `valid` depending on whether the pointer representing the choice member is NULL or pointing to some data.

  **SET** and **SEQUENCE** a fourth element, a list of pairs, is returned, where the pairs are built from the member name and an identifier that is either `valid` or `void`

  **SET OF** and **SEQUENCE OF** the number of items is returned as element number 3.

**snacc get *path*** returns the value of the subtree pointed to by *path*. The value returned is a simple string for simple types, and a hierarchical structure (in Tcl that is a list of lists) otherwise.

**NULL** the empty string is returned.

**BOOLEAN** the value is returned as `TRUE` or `FALSE`.

**INTEGER** the numeric value is returned, even if it has been assigned a name.

**ENUMERATED** the symbolic value is returned. The numeric values are inaccessible through the Tcl interface. If the object happens to contain an illegal numeric value, an error is returned and `errorCode` is set to `SNACC ILLENUM`.

**REAL** the value is returned as formatted by sprintf (..., "%g", ...), except for the special values PLUS-INFINITY and MINUS-INFINITY which are returned as `+inf` and `-inf`, respectively.

**BIT STRING** a string, consisting solely of `0' and `1', is returned.

**OCTET STRING** the binary string is returned as is, except for the unavoidable NUL-escape described above.

**OBJECT IDENTIFIER** the value is returned as a list of numbers.

**CHOICE** is returned as a pair, the choice member chosen and its value.

**SET** and **SEQUENCE** are returned as a list of pairs of member name and value. Absent OPTIONAL members are left out from the list.

**SET OF** and **SEQUENCE OF** are returned as a list of values.

**snacc set *path value*** sets the subtree identified by *path* to *value*. The value must be of the form

**NULL** the only legal value is the empty string. otherwise, an error is returned and `errorCode` is set to `SNACC ILLNULL`.

**BOOLEAN** any value that is accepted by Tcl_GetBoolean(3) is fine.

**INTEGER** both the numeric (as accepted by Tcl_GetInt(3)) and the symbolic values are allowed.

**ENUMERATED** any value must be specified by its name. If an illegal name is given, an error is returned and `errorCode` is set to `SNACC ILLENUM`.

**REAL** the special values PLUS-INFINITY and MINUS-INFINITY have to be given as `+inf` and `-inf`, respectively. All other values may be specified in any format accepted by Tcl_GetDouble(3).

**BIT STRING** a string that must consist of `0' and `1' only has to be given. otherwise, an error is returned and `errorCode` is set to `SNACC ILLBIT`.

**OCTET STRING** due to the NUL-escapes necessary, any string where a backslash is followed by either another backslash or a `0' digit is legal. Improper use of the escape character leads to an error and `errorCode` will be set to `SNACC ILLESC`.

**OBJECT IDENTIFIER** the value has to be specified as a list of numbers. if the arc has less than 2 or more than 10 elements, an error is returned and `errorCode` is set to `SNACC ILLARC <2` or `SNACC ILLARC >10`, respectively.

**CHOICE** the value expected is a pair, the choice member chosen and its value. if an illegal member is specified, an error is returned and `errorCode` is set to `SNACC ILLCHOICE`.

**SET** and **SEQUENCE** the value has got to be a list of pairs of member name and value. Any member may be specified at most once. All mandatory members must be present. Failure to do so will result in an error and `errorCode` to be set to `SNACC DUPMEMB` or `SNACC MISSMAND`, respectively. All optional members not listed in the value will be deallocated.

**SET OF** and **SEQUENCE OF** the whole list is replaced with the specified value that has to be a proper Tcl list.

**snacc unset *path*** unsets the subtree pointed to by *path*. Only OPTIONAL members of SET and SEQUENCE types and list elements of SEQ OF and SEQUENCE OF may be unset. If you try to unset a mandatory SET or SEQUENCE member, an error is returned and `errorCode` is set to `SNACC MANDMEMB`.

I did not follow Tk's example where one has to set widget commands to `{}` to delete them. This method would have the drawback that one could not distinguish between an empty and a non-existing octet string (in C that would be "" vs. NULL).

The value returned by `snacc get` may be very long, `snacc get file0` returns the contents of the whole file!

## 9.3 Examples

The following example session shall illustrate the `snacc` commands usage. It assumes that the editor example files `edex0.asn1` and `edex1.asn1` (see appendix C on page 174) have been compiled into a binary that has been linked with the necessary libraries.

The notation used is as in the Tcl book [9], i.e. `⇒' indicates a normal return value and `∅' indicates an error with the error message set in *oblique typeface*.

A look at the types available:

```
   snacc types
⇒ {EdEx-Simple Hand} {EdEx-Structured StructuredChoice}
   {EdEx-Structured Coordinate} {EdEx-Structured CoordinateSeq}
   {EdEx-Structured RGBColor} {EdEx-Structured Simple}
   {EdEx-Simple File} {EdEx-Simple RainbowColor} {EdEx-Structured
   DirectorySetOf} {EdEx-Structured Various} {EdEx-Structured
   File1} {EdEx-Structured CoordinateSeq1} {EdEx-Structured
   Directory} {EdEx-Structured Structured} {EdEx-Simple
   DayOfTheWeek}
```

Create a file (without filename):

```
   set file [snacc create {EdEx-Structured Structured}]
⇒ file0
```

The string returned is the file handle. It is used as the first snaccpath component in successive calls.

Look at the file's type:

```
   snacc info $file
⇒ {EdEx-Structured Structured} sub SET {{coord valid} {color
   valid}}
```

The file's type is a SET with the name `Structured' in module `EdEx-Structured' (it is defined in file `edex1.asn1` (see page 174)). The `sub' tells us that the type has not been marked as a PDU. The SET has the components `coord' and `color', both are present (they are not OPTIONAL, i.e. mandatory).

Look at a component's type:

```
   snacc info "$file color"
⇒ {EdEx-Structured StructuredChoice} sub CHOICE rainbow valid
```

Snacc has generated the type name `StructuredChoice' for this type, this name was not defined in the `.asn1` file. The CHOICE object currently is set to `rainbow'. A CHOICE component is always present (CHOICE components may not be OPTIONAL), the `valid' is just for completeness.

Ask for the CHOICE's generic type information:

```
   snacc type {EdEx-Structured StructuredChoice}
⇒ {EdEx-Structured StructuredChoice} sub CHOICE {{rainbow
   {EdEx-Simple RainbowColor} sub INTEGER} {rgb {EdEx-Structured
   RGBColor} sub SEQUENCE}}
```

The CHOICE type has two possible components, `rainbow', an INTEGER and `rgb', a SEQUENCE.

Look at the INTEGER's type information:

```
   snacc type {EdEx-Simple RainbowColor}
⇒ {EdEx-Simple RainbowColor} sub INTEGER {{red 0} {orange 1}
   {yellow 2} {green 3} {blue 4} {indigo 5} {violet 6}}
```

The type has got named values.

Access the file contents:

```
   snacc get $file
⇒ {coord {cartesian {{x 0} {y 0}}}} {color {rainbow 977768}}
```

The color component contains garbage. Change that:

```
   snacc set "$file color rainbow" green
⇒
   snacc get "$file color"
⇒ rainbow 3
```

Change it again, select the CHOICE's other component type, `rgb', and set its `red' component:

```
   snacc set "$file color rgb red" 256
⇒
```

Changing a CHOICE component selection work only for write access, on read access this is not possible:

```
   snacc get "$file color rainbow"
ø  snacc get:  illegal component "rainbow" in path
   snacc get "$file color rgb"
⇒ {red 256} {green 544501616} {blue 1814045815}
```

Upon setting a SET or SEQUENCE type, all mandatory members have to be specified:

```
   snacc set "$file color rgb" {{green 0} {blue 0}}
ø  mandatory member "red" is missing in list
   snacc set "$file color rgb" {{red 0} {green 256} {blue 0}}
⇒
   snacc get "$file color"
⇒ rgb {{red 0} {green 256} {blue 0}}
```

Finish up:

```
   snacc close $file
⇒
   snacc get $file
ø  snacc get:  no file named "file0"
```

## 9.4 Implementation

The Tcl interface is implemented in `.../c++-lib/inc/tcl-if.h` and `.../c++-lib/src/tcl-if.C`. It gets initialized with the help of `.../c++-lib/inc/init.h` and `.../c++-lib/src/tkAppInit.c`.

The `snacc` commands implementation is pretty straight forward: check the arguments, call a metacode function to perform an action and return the result, which may indicate success or an error.

Care has been taken to check the return codes of all system calls and to set Tcl's `errorCode` variable in case any system call returns an error.

The file `tkAppInit.c` contains the function that introduces the `snacc` Tcl command to the Tcl interpreter. The path that leads to the function's invocation is a little tricky and is described in section 10.3, “Building Your Own Editor”.

## 9.5 Setup for the Tcl Code Generator

To compile Snacc with the Tcl interface code generator, you have got to fulfill the following conditions:

- the configure script must be able to find `tclsh` and the Tcl/Tk libraries
- the preprocessor switches NO_META and NO_TCL in `.../policy.h` must not be set

## 9.6 Deficiencies

- Values defined in the ASN.1 files currently are inaccessible. Adding access functions to the metacode and Tcl interface is rather trivial: build an array of elements that hold a variable's name as a character string and an AsnType * that points to the C++ variable. a[i].val–>_getdesc() would return a pointer to the variable's type description.

  (First you should fix snacc's value parser as currently it lets some values silently vanish, for example the victory in `edex0.asn1` that you can find in appendix C on page 174.)

- The Tcl interface does not provide symbolic object identifiers. Mapping numeric to symbolic oids is a task that is difficult to get right since snacc translates
  anOidVal OBJECT IDENTIFIER ::= { 1 2 foo(3) }
  and
  anOidVal OBJECT IDENTIFIER ::= { 1 2 3 }
  foo INTEGER ::= 3
  into identical C++ code, but translating the second anOidVal into { 1 2 foo } may in fact violate foo's semantics.

# Chapter 10

# SnaccEd, the Snacc Editor

SnaccEd is a simple graphical editor for BER encoded files. A set of ASN.1 files describes one or more hierarchical datastructures that can be displayed as an n-ary non-circular graph, in other words: a tree.

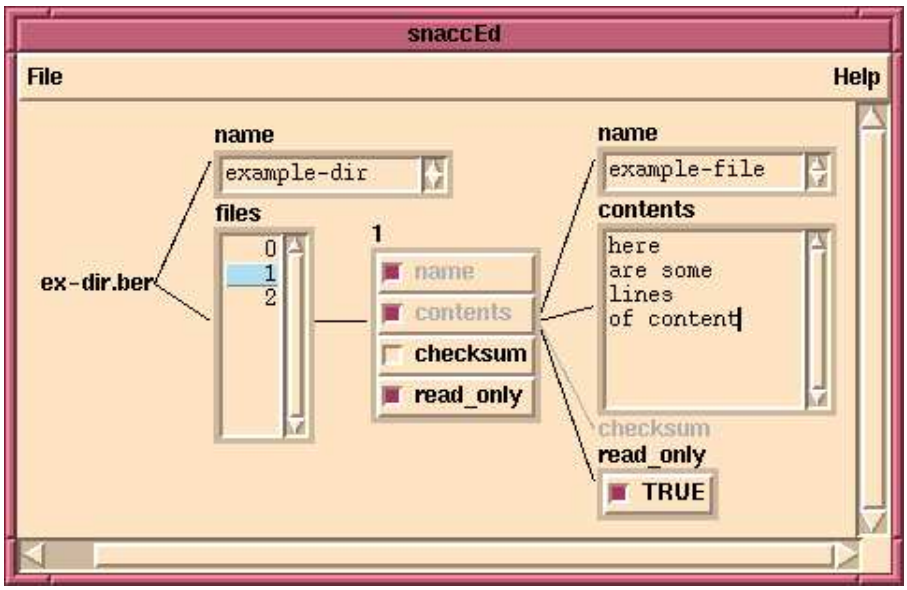


Figure 10.1: An example screen shot

SnaccEd is combined from

- the usual Snacc BER encode and decode functions
- the metacode (described in chapter 8 starting on page 107)
- the Tcl library

- the Snacc Tcl interface (described in chapter 9 starting on page 121)
- the Tk widget set
- a freeware tree widget (another Tcl extension, implemented in C++)
- a Tcl script that glues all those parts together

All items except for the Tcl script are compiled into an executable, the *snaccwish*. The Tk interpreter has the name `wish`, for "windowing shell", and consequently, I name the program that results from linking Snacc with Tk `snaccwish`. For every individual set of ASN.1 files, a different snaccwish has to be made, because every snaccwish contains the specialized encode and decode routines for the ASN.1 files' types.

This `snaccwish` is a Tcl interpreter that has the additional commands of the Tk widget set, the tree widget and the Snacc interface built in. This interpreter reads the `snacced` script that implements the graphical Snacc editor. I will henceforth refer to the script as `snacced` and to the interpreter as `snaccwish`. You may name your script and the shell binary differently, just make sure that the script calls the correct binary!.

Snacced can be called with various arguments, see the manual page for details.

The `snacced` script is only the most visible entry point, other scripts will be read using Tcl's autoloading mechanism.

The Tcl script is (or can be) always the same. It uses the `snacc` command to learn about the ASN.1 modules, types and PDUs.

Since the BER format has not got any magic number or similar concept, the Snacc routines in general cannot identify the ASN.1 type contained in a BER encoded string of octets. As a consequence, one has to choose not only the file name but the ASN.1 type as well when one opens or creates a file (see figure 10.2 on page 133 for an example).

One can then examine and manipulate the file's structure and contents.

## 10.1 Manipulating the Display

This section describes the pointer[1] operations that change the amount of information to be shown. (To change the file's contents, the node's content window has got be opened.)

The file is displayed by means of a tree widget. Only a part of the full hierarchy is shown. The subtree's root is at the left side. The function of the pointer buttons[2] when clicking on *node names* is as follows:

**button 1** adds or deletes the node's subnodes to or from the display, respectively. (Except for SET OF and SEQUENCE OF types, where with button 2 you have got

[1] My pointer device is a mouse, but yours may be a trackball, a tablet, a joystick or something else.

[2] I will refer to the buttons but their number, not their position. I could refer to button 1 as the right button, but this might confuse you as your button 1 may in fact be on the left hand side.

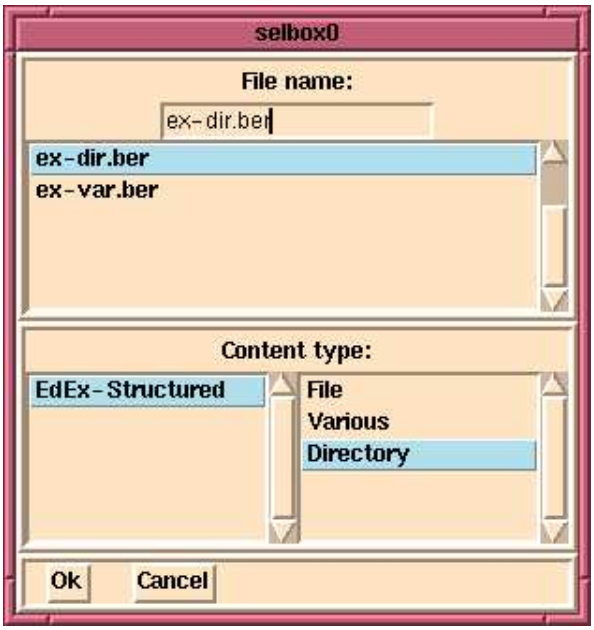


Figure 10.2: The file and content type selection box

to open the node content editor, a list widget, and have to toggle the display of individual elements by clicking on their index numbers. This is explained at the end of section 10.2 on page 136.)

- For nodes that have subnodes being shown, the subtree gets hidden.
- Otherwise, the node's immediate descendents are added to the display.

**button 2** opens or closes the node, where “closed” means that only the nodes name is being shown, and “open” means that an additional window showing the node's contents it put under the node's name. This content window is explained in the next section.

**button 3** adds or deletes the node's parent to or from the display, respectively.

- For nodes where the parent is displayed, all parents and all siblings with their subtrees will get hidden.
- Otherwise, the parent is added to the display.

Pressing and holding button 2 on a free space, the display can be dragged by moving the pointer.

## 10.2 The Content Window

The content window that may be opened beneath the node's name looks and behaves different for every content type. An example for every ASN.1 simple type is shown in

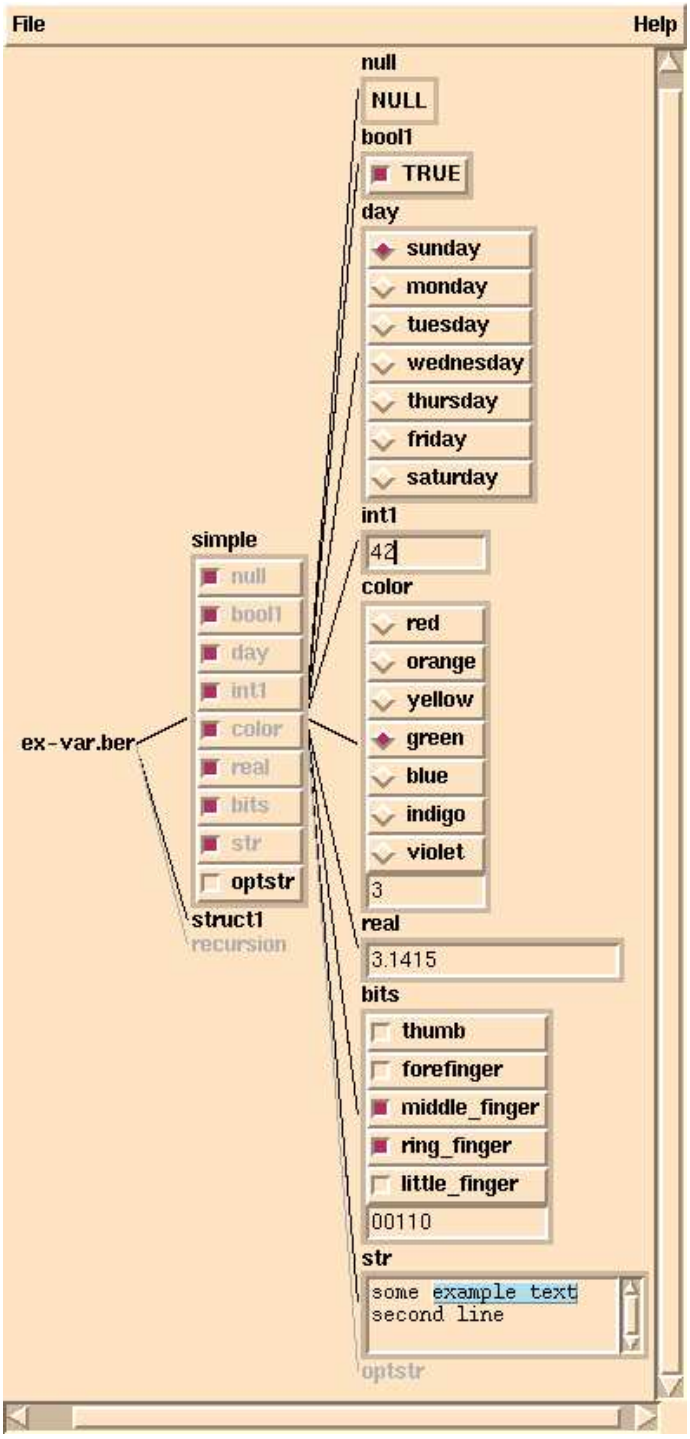


Figure 10.3: Content editors for ASN.1 simple types

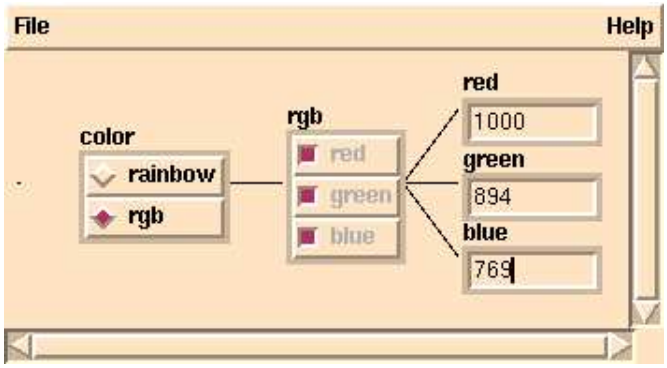


Figure 10.4: Content editors for ASN.1 structured types


Figure 10.5: Popup for import/export of OCTET STRING contents (based on the example displayed as figure 10.1 on page 131)

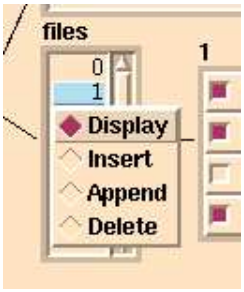


Figure 10.6: Popup for action selection for SET OF and SEQUENCE OF types (based on the example displayed as figure 10.1 on page 131)

figure 10.3 on page 134. The ASN.1 input for the example can be found in appendix C on page 174.

- The NULL type has only one value that cannot be changed.
- Values of BOOLEAN type are displayed as a toggle button.
- For the ENUMERATED type, SnaccEd displays a list of radio buttons listing the values' names. (The numerical values are not shown.)
- INTEGER values are displayed using an entry widget where the numeric value can be seen and changed. The entry widget's binding have been changed to allow the input of "–" and decimal digits only in addition to the usual control functions (procedure `int_entry_bindings`). Similar to the ENUMERATED type, values can be given a name; the list is displayed as above.
- Individial names in a BIT STRING may be named. SnaccEd displays a list of buttons identifying those bits along with their name. Clicking on one of those buttons toggles the bit's value.
  The bit string is displayed and can be edited in its binary representation in an entry widget below the names. The entry widget's binding have been changed to allow the input of "0" and "1" only in addition to the usual control functions (procedure `bit_string_entry_bindings`).
- OCTET STRINGs and derived types are displayed in a text widget. Since Tcl cannot handle strings containing NUL bytes, NUL bytes are displayed as the two character combination "\0" and backslashes are duplicated, "\\". Button 3 pops up a small menu that allows you to load or save the octet string from or to an external file, respectively (figure 10.5 on page 135). The X text selection to copy text between the text widget and e.g. an xterm can be used as well.
- CHOICE types allow exactly one of their subtypes to be valid and therefore are displayed as a list of radio buttons. Clicking on a button deletes the old choice and allocates the new one. See the "color" in figure 10.4 on page 10.4.
- The SET and SEQUENCE types' elements are displayed in a list of buttons, and optional elements may be added and deleted by clicking on their buttons. Mandatory members do not respond to button clicks and are greyed out. Examples: the list element #1 right in the middle of figure 10.1 on page 131 or the "rgb" in figure 10.4 on page 135.
- SnaccEd visualizes the types SET OF and SEQUENCE OF in a list widget. The widget shows the elements' ordinal numbers, the elements themselves are shown in individual widgets to the right. Button 3 in the list widget brings up a small menu where you can choose the action to perform when (with button 1) you click on a list element (figure 10.6 on page 135):
  - toggle the display of an element
  - insert a new element
  - append a new element
  - delete an element

The cursor shape changes and reminds you of your chosen action. An example of a list widget is "files" in figure 10.1 on page 131.

Some content editors can be resized: move the pointer to the content editor's frame. Where the cursor shape changes to a "bottom_right_corner", press button 1 and drag the frame.

## 10.3 Building Your Own Editor

There isn't much to be done to get an editor that understands your BER encoded files. This chapter's example can be found in `.../tcl-example/`.

- Make sure your Snacc compiler has be configured to support the Tcl code. This can be verified by calling `` `snacc -h | grep tcl` ``; if the output is empty, the configuration script was unable to find Tcl/Tk.
- The Tcl/Tk libraries must have been compiled with with gcc in order to use its main() function. Otherwise, the constructors and destructors of static variables may not be called. I have added some code at the end of `.../.../c++-lib/src/tcl-if.C` that checks for this condition.
  Tcl's default is not to use gcc but cc where present, but you can compile Tcl/Tk by calling `env CC='gcc -traditional' ./configure` instead of `./configure`.
- Install the tree widget. SnaccEd works with tree-3.6. (tree-3.6.2 requires another Tcl extension, itcl, but to avoid complicating matters any further, SnaccEd sticks to the simpler version.) When `.../configure` found the tree widget libraries, the macro definition for `TREELIBS` in `.../makehead` will have been set to `-ltktree -lOS`.
- In your makefile, extend `snacc`'s list of arguments with the `-tcl` option followed by the list of PDUs (that is an additional *two* arguments).
- Put `modules.C` into the list of files to be compiled.
- Compile with `-DTCL`.
- To link, replace `-lasn1c++` against `-lasn1tcl` and add `$(TREELIBS)` to the list of libraries. (You may have to add a `-L` option as well.)
- Call `$(TCL_LIB)/make-snacced $(SNACCED) $(WISH) $(TCL_LIB)`, where `$(TCL_LIB)` is either `${prefix}/lib/snacc/tcl/` (the place where the Tcl library files got installed by `make install`) or `.../tcl-lib/` (in case you haven't called `make install` yet), and `$(WISH)` is the name of the wish executable you just compiled. The `make-snacced` script generates a small Tcl script, `$(SNACCED)`, that executes your wish executable, extends the Tcl variable `auto_path` to include your `$(TCL_LIB)` und through the auto loading mechanism calls the Tcl library routine `snacced` to start the Snacc editor.

Make sure you get the `.../tcl-example/` working before you despair of your own set of files. Take `.../tcl-example/makefile` as a guide.

Your can use the `.h` and `.C` file resulting from calling `snacc -tcl ...` for both the editor and your other uses. Just remember: if you compile with `-DTCL`, link against `libasn1tcl.a`; if you want to disable the metacode and tcl interface, compile with `-DMETA=0` and link against `libasn1c++.a`.

Make sure that you don't link with your old main()[3]. The generated file `modules.C` contains the line
static int (*dummy)(Tcl_Interp *) = Tcl_AppInit;
that forces `libasn1tcl.a(tkAppInit.o)` to be linked. `.../c++-lib/src/tkAppInit.c` in turn contains the lines
extern int main();
int *tclDummyMainPtr = (int *)main;
that force the main function in the Tk library to be linked.

Tcl_AppInit() calls Snacc_Init() that is defined in `.../c++-lib/src/tcl-if.C`. Snacc_Init() installs the `snacc` command.

## 10.4 Implementation

The Tcl scripts that implement the editor can be found in `.../tcl-lib/` and, after installation, in `${prefix}/lib/snacc/tcl/`.

You are free to change the Tcl script(s), for example to display some data types in a more appropiate manner. Octet strings may be user readable but often are not, pictures and audio data come to mind.

If add procedures or `.tcl` files, you have got to rebuild Tcl's autoloading index. This best done by adding the files to the `TCLFILES.dist` list in `.../tcl-lib/makefile` and running `make` again.

The first SnaccEd was able to handle only one file at a time. To enable the editor to handle several files simultaneously required the following steps:

- instead of using the default toplevel widget `.` (dot), open a toplevel widget for every file. The toplevel widgets get names `.file0`, `.file1`, ...The same name without the leading dot is used as a global array variable to hold miscellaneous pieces of information about the file. You can see how the names are generated in the code example on page **??**. The name of this variable is given to many procedures in the `fileref` parameter.
- identify global variables. Those were
  - the name of the file handle
  - the names of some widgets, namely the toplevel, the menubar, the canvas and the tree widget.

[3]Or make sure your main() behaves similar to the Tk libraries'

Other global variables can be left untouched: the help text, the list of PDU types. This information is the same for all the files a snaccwish can handle.

Since the file and type selection box, the help text and the dialog boxes are modal, only one instance is needed and they can have the same names for every file opened by the editor.

The editor displays only a portion of the ASN.1 file. The Snacc editor keeps the displayed portions of the ASN.1 file in two similar data structures.

The contents of an ASN.1 file is accessed by calling the `snacc` command with a *`path`* that identifies the requested data portion.

Every ASN.1 file is displayed using one toplevel widget. This toplevel widget is a frame for a number of subwidgets:

- a menubar
- a canvas
- two scrollbars, one vertical, one horizontal, to select the visible part of a canvas that has grown too large for the frame.

The menubar contains two buttons, one for the usual file related commands, and a help button.

The canvas is the main arena. Its subwidgets are the tree widget and all the canvas items that make up the nodes and edges. The tree widget computes the positions of the canvas items and moves them in place.

The contents of an ASN.1 file can be seen as a tree (the data structure may by recursive using CHOICE types or OPTIONAL components, and a PDU may contain instances of a type that contain other instances of the same type (see figure 10.7 on page 142 for an example), but as ASN.1 has no pointers, cycles are impossible). To display this tree, it is mirrored in a number of Tcl data structures:

- The *snaccpath* is the 1:1 representation of the PDU's structure. This is what in chapter 9 is always referred to as "*path*" argument to most `snacc` subcommands. The snaccpath is a proper Tcl list.
- The *treepath* is very similar to the snaccpath. Its structure is the same as the snaccpath's, but its syntax and a few elements are different:
  - The components in a snaccpath are separated by " " (space), in a treepath they are separated by "/" (slash). This difference is not strictly necessary, but it helps to detect errors in argument passing as the `snacc` commands will never accept any treepath for their path arguments.
  - In a snaccpath, the elements of SET OF and SEQUENCE OF types are identified by their index. In a treepath, another numeric id is used instead. The reason for this becomes clear when we have a look at where the treepath is used and what would have to be done if the elements' list indices were used in the treepaths.

The treepath is used in a number of places, for widget and variable names and for canvas item tags, all detailed in the below bulleted items.

When an element of a SET OF or SEQUENCE OF type is deleted, the snaccpath's indices for the deleted element's successors have to be decremented to point to the same item; when an element is inserted, those indices need to be incremented. As a consequence, the widget and variable names and the canvas item tags of all elements that follow the one element that has been deleted or inserted would have to be adjusted and all the names and tags of their descendants. Even if these names and tags could easily be changed (they cannot), it would still be an enormous amount of work and the slow Tcl interpreter could need some seconds to complete this task. This enormous labour can be avoided by introducing a table lookup:

Every node of a SET OF or SEQUENCE OF type gets an idlist (identifier list). This idlist is a Tcl list, its length is the same as there are elements in the ASN.1 data object. Every idlist element corresponds to an element of the data object. Whenever an element is deleted from the data object, the corresponding id from the idlist is removed as well; insertions are likewise performed in both the data object and the idlist. The idlist contains numbers, zero for data objects that are not visually displayed on the canvas and locally unique non-zero numbers otherwise.

When a data object is identified through its treepath, the id is extracted and the id's position is sought in the idlist. The id's position in the idlist is the element's index for the snaccpath.

The treepath is used to build the names of widgets that display a PDU's structure and content portions.

- The node labels and lines for the edges are canvas items, no full fledged widgets. Canvas items can be given tags for identification purposes; the tags of an item are an ordered Tcl list. Canvas items have a locally unique id, but as different items can have the same tag, item groups can be identified.

  Since all tags form an ordered Tcl list, individual items can be addressed:
  `[lindex [$canvas gettags $id] $index]`
  SnaccEd uses this mechanism to translate button clicks into paths: when a canvas item is clicked at, the canvas makes this item "current" and
  `[$canvas find withtag current]`
  returns the item's id. The id is then used as described above to retrieve the tag list.

  The canvas line items that are used as edges get no tags.

  The canvas text items that make up the node labels and the canvas window items that contain the content editors get three tags. The three tags are ordered from most general to most specific:

  0. For node labels this tag has the form *validity*-label. The validity is either "valid" or "void". Absent OPTIONAL components are "void". Active node labels get the tag "valid-label". In the procedure `new_file` this tag is used to bind the three pointer button events to the callback procedures `prune_or_add_children`, `toggle_editor` and `set_or_add_root`, respectively.

For content editors this tag is simply “edit”, because content editors can only be opened for valid nodes and therefore the validity would be redundant.

1. This tag is the treepath. It is the same for all canvas items for this node: the label and possibly the content editor. This is the tag that is given to the tree widget. The tree widget handles all canvas items with the same tag as a group: it uses their bounding box to calculate the tree layout and it keeps the relative distances of the group's items so that their internal layout persists any change in the tree's layout.
2. This tag is a combination of the other two tags: it is the treepath, a colon and either “label” or “edit”. This tag is the most specific and it is used to address the individual canvas item, for example to check for a content editors existence. No two items have the same value of this tag.

- Content editors are not simple canvas items. They are build from one or more widgets and this widget tree is put into a canvas window item. The widgets have names of the form `$canvas.edit$treepath`. The leading $canvas is the name of the canvas widget. Widget names starting with that name are descendants of the canvas, here they are children. The trailing $treepath does not contain any dots and therefore Tk understands edit$treepath as a single node in the widget tree.

- Most of the content editors modify a global variable, for example the entry widgets for INTEGER types or the radiobuttons for ENUMERATED types. The variable's name is the simple composition `var:$treepath` that guarantees its uniqueness.

  SET and SEQUENCE types need a variable for each of their components: the component's name gets tacked to the end which yields `var:$treepath:$name`.

  Named bits of BIT STRING types get similar variables, the bit value is put after the second colon. The bits' toggle buttons operate on these variables.

When button 2 is clicked on a node label, the procedure `toggle_editor` gets called:

- checks whether the editor for the current node already exists.

  If it does, the editor is deleted.

  Otherwise, the editor is opened by creating a frame widget that is filled with an appropiate set of subwidgets. The editor is supplied with the corresponding content portion from the ASN.1 file and a number of event bindings is installed that let the user modify the contents. The frame is placed at the right place below the node name and the tree widget is called to adjust the layout. Most of the changes to nodes containing simple ASN.1 types are detected using Tcl's trace mechanism. For example, the entry widget for an INTEGER modifies a global variable. The trace procedure that gets called upon every modification computes the snaccpath from the variable's name and modifies the ASN.1 file accordingly. The only ASN.1 simple type that makes an exception to the rule is OCTET STRING. Text widgets do not modify global variables and even if they

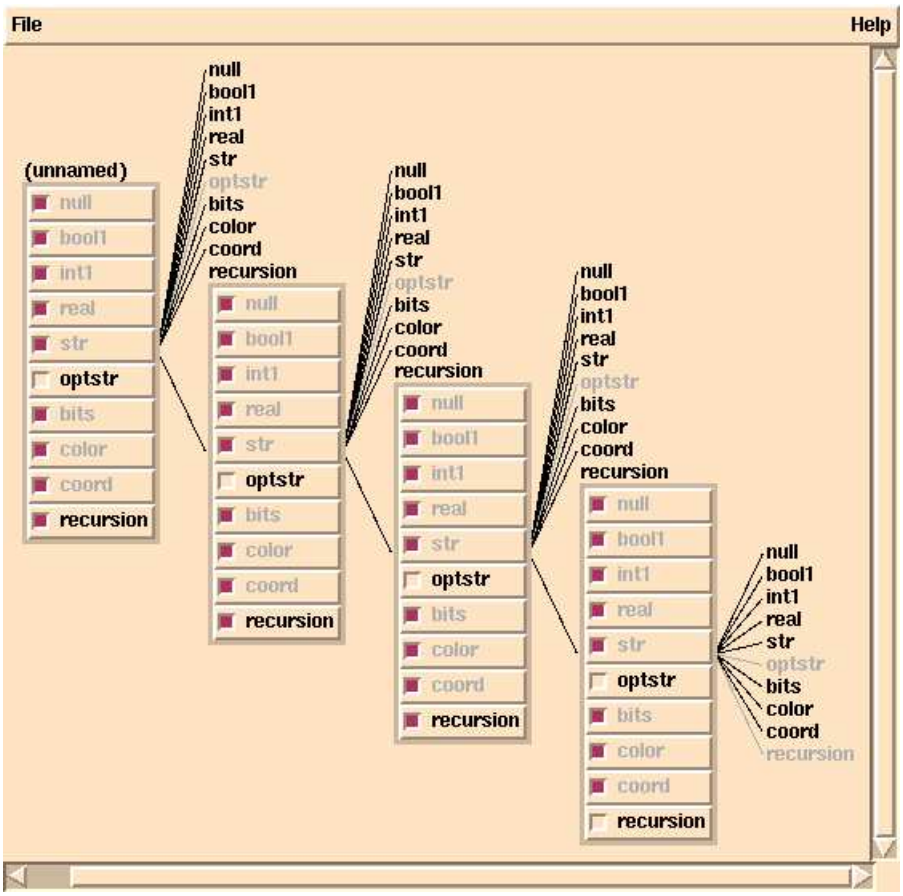


Figure 10.7: `Recursive' data structures

did, copying the string back to the Snacc object for every key stroke is a waste of CPU time. The text widget's contents is written back to the octet string when the widget receives a leave window event or, since Tk uses explicit focus (click-to-focus instead of focus-follows-pointer) and thus the widget's contents may be changed even after the pointer has left the window, it receives a focus out event.

The structured ASN.1 types are modified explicitly, i.e. through button clicks. The components are modified using their natural GUI counterparts.

- OPTIONAL members of SET and SEQUENCE types are allocated and deallcated by clicking on a checkbutton.
- CHOICE members are selected by clicking on a radiobutton.
- SET OF and SEQUENCE OF types are implemented as list and SnaccEd displays them using a list widget. (This is not actually true. Since the listbox widget allows only single selection or selection of multiple but contiguous entries, I replaced it with a text widget and with some event bindings and a tag for the selected items made it behave like a listbox widget with multiple selection. A patch to the Tk 3.6 code that implements non-contiguous selections exists, but I didn't want to enforce the trouble of getting, patching and installing the Tk library again on anyone.)

set_or_add_root:

When button 3 has been clicked on a node label, the callback procedure `set_or_add_root` will be called.

- The procedure checks whether the node that has been clicked at is the file's toplevel node. In this case nothing can be done because there is no parent that could be shown or hidden.
- Otherwise, the routine checks whether the clicked node is currently displayed as the subtree's root. In this case both the node's parent and grand parent are calculated. The grand parent is not displayed, it merely corresponds to the tree widgets invisible root. The parent is displayed by adding the node to the invisible grand parent. The old root node and its siblings are aded by calling the `ed_expand` procedure. `ed_expand` in turn calls `ed_addnode` for all the parent's children. `ed_addnode` checks whether the node is already present on the canvas. For the old root node this check yields true and the routine simply moves the node and all its descendants to below the new root node. All other nodes are created as usually done when clicking on button 1.
- Otherwise, the node that has been clicked is to be shown as root. This is simple: just tell the tree widget to display the node as root. The tree widget removes everything else that does not belong to the subtree and calls the remove callback procedures for all these items. The remove callback procedure is installed when a content editor is opened and is used to destroy the content editor frame widget and its subwidgets.

selbox:

The file-and-content-type-selection-box (short: selbox) serves different purposes. The selbox contains three parts: a file name selection, a content type selection and a button row. One of the two selection parts can be disabled (it will not even been shown). In figure 10.2 on page 133 both parts are visible.

Both selections are necessary if a user wants to open a file. The user is the only one to know which of the PDU types is contained in the file.

Only the file selection is necessary to implement the usual "Save As…" functionality where the content type is already known.

Only the content type selection is needed when the program wants the user to create a new file without giving it an external file name. SnaccEd currently has no such function. Instead, if a user upon opening a file does not select a file name, an internal file without an external file name gets created.

The selection box is implemented in the file `.../tcl-lib/selbox.tcl`. In this file, every procedure name starts with the prefix `selbox_` (except for the main entry point, `selbox`).

If the selbox was made non-modal, it would not break, because each individual selbox widget gets its own widget tree and all its status variables are put into a uniquely named array variable. The code that generates the names is similar to the example on page **??**.

# Chapter 11

# IDL Code Generation

## 11.1 Introduction

*Under construction*

# Chapter 12

# Type Tables

Type tables are a flexible and compact way of dealing with ASN.1. The type table data structure is included in the appendix. It was defined in ASN.1 to provide a good storage format for the tables.

When snacc produces a type table it includes the useful types module as well, if one was specified. If you are really trying to limit the size of your type tables, put the only useful types that you need in your ASN.1 module and compile it without using the useful types module.

A generic buffer type (à la ISODE and XDR) was defined to allow type table driven routines to read from a wide variety of buffer formats. Currently slightly modified versions of the ExpBuf and the SBuf are provided. It shouldn't be too hard for you to add support for your own buffer formats. The generic buffers, GenBufs are described in more detail in a following section.

The general procedure for using type tables is to:

1. Use snacc to compile your ASN.1 modules into a type table.
2. Use `mkchdr` (make C header, not make cheddar) with the type table to produce a friendly C type description of the types in the type table.
3. Load the type table during runtime and use it to configure the table encode, decode and other routines.

Step two, making the C header file is not necessary but will make dealing with the value easier and more type safe. Internally the table driven encoders and decoders know nothing of these header file and treat the types in a uniform, generic manner. This requires the encoders and decoders to make assumptions about the way C represents strucutures and values. Look in the `.../c-lib/src/tbl-enc.c` and `.../c-lib/src/tbl-dec.c` files to see how this generic data structure is manipulated.

On the down side, the compiler directives do not affect the data structures generated by mkchdr and the generated C type definitions will generally be different from those generated by the C backend. This can be fixed, but time was lacking. Type tables also

do not support ANY DEFINED BY types. Someone could fix this without too much difficulty. Only a C type table library is provided. I didn't have time to deal with the complexities of creating C++ objects in a generic way.

Currently the type tables are lacking subtyping information. It is available in snacc's main parse tree but I didn't have time to add it to the tables. If you want to add it, take the subtype related data structures (in ASN.1) from asn1module.asn1 (quite a few), remove all the cruft pertaining to linking and error checking etc, and add it to the type table type definitions. Then change the `.../compiler/core/gen-tbls.c` file to take the subtype information from the parse tree and put it into the type table. See the appendix or `.../asn1specs/` for the ASN.1 definitions of the parse tree and type tables.

The parse tree itself was defined in ASN.1 so it could be the table format. The extra complexity required for linking and error checking made this very difficult. Cycles in the data structure and the many links between the data elements made encoding in BER difficult. [Maybe ASN.1 needs a type refernce type (i.e. pointer)].

## 12.1 How Type Table See Values

As mentioned in the last section, table driven encoding, decoding, printing etc. routines see your values in a generic way. They do not have abstract syntax specific header files like those created by `mkchdr`.

The basic idea is that all of the standard primitive and list (SEQUENCE OF and SET OF) types are used and some regular rules are used for allocating and laying out structs for SEQUENCE, SET and CHOICE types.

```
/* values from table driver routines' point of view */
typedef void AVal;

/* SEQUENCE and SET type use AStructVal */
typedef AVal *AStructVal;  /* an array of AVal ptrs */
```

For SETs and SEQUENCEs, the AStructVal type is used. Its is basically an array of pointers, one for each component of the SET/SEQUENCE. Every component is referenced by pointer to simplify allocations. OPTIONAL or DEFAULT components may be NULL. For example, the type:

```
Foo ::= SEQUENCE { a INTEGER, b BOOLEAN, c OCTET STRING }
```

would be represented as an array of three pointers. The first pointer would point to an AsnInt value, the second would point to an AsnBool value, and the third would point to an AsnOcts value. `mkchdr` would give you the following C typedef for Foo:

```
typedef struct Foo
{
```

```
    AsnInt *a;
    AsnBool *b;
    AsnOcts *c;
} Foo;

/* Internal representation for a CHOICE type */
typedef struct AChoiceVal
{
   enum { achoiceval_notused } choiceId;
   AVal *val;
} AChoiceVal;
```

A CHOICE type is represented in a way similar to the C backend's output. That is, an enum to hold the id of the CHOICE component that is present and a pointer to the component itself. For example, the type:

```
Bar ::= CHOICE { a INTEGER, b BOOLEAN, c OCTET STRING }
```

would internally be represented as AChoiceVal type. However, `mkchdr` would give you the following:

```
typedef struct Bar
{
    enum
    {
        a_ID = 0,
        b_ID = 1,
        c_ID = 2
    } choiceId;
    union
    {
        AsnInt *a;
        AsnBool *b;
        AsnOcts *c;
    } a;
} Bar;
```

## 12.2 Type Table Utilities

There are a bunch of useful routines in `.../c-lib/src/tbl*.c`. Look at the source code in `.../tbl-tools/*/` and `.../tbl-example/` to see how to use some of them.

The LoadTblFile will decode a type table from a given file. Notice that its definition of the TBL data structure has been augmented to simplify encoding and decoding operations. (Look at the patch in `.../c-lib/tbl.h.patch` that is applied through the makefile automatically.) The compiler uses unmodified `tbl.h` and `tbl.c` files.

I don't have time to document these routines. Look through the table tools, examples and library code. Their usage should be fairly obvious.

## 12.3 Type Table Tools

The `.../tbl-tools/` directory contains three tools, `mkchdr`, `pval` and `ptbl`. These are described in the following sections.

### 12.3.1 Making C Header Files with mkchdr

`mkchdr` produces a C header file from a type table. This header file shows the representation of the types that the table tools will expect or return for the types in the given type table.

The main use is to provide you with an easy to use definition of the ASN.1 types C representation. You do not need to use `mkchdr` but it is definately recommended. Note that the table routines could have used an even more generic data structure to represent values (e.g. ISODE's Presentation Elements). If you have worked with these, you know that they are cumbersome.

Its synopsis is:

```
mkchdr <tbl-file> [output-file]
```

If the output file is omitted, the header file is printed to stdout.

Here is an example of the output. Given the table that has the following ASN.1 module in it:

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

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

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

```
EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER

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

END
```

`mkchdr` will produce:

```
typedef AsnInt EmployeeNumber;

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

typedef IA5String Date;

typedef struct ChildInformation
{
    Name *field0;
    Date *dateOfBirth;
} ChildInformation;

typedef AsnList PersonnelRecordSeqOf;

typedef struct PersonnelRecord
{
    Name *field0;
    IA5String *title;
    EmployeeNumber *field1;
    Date *dateOfHire;
    Name *nameOfSpouse;
    PersonnelRecordSeqOf *children;
} PersonnelRecord;
```

### 12.3.2 Printing Tables with ptbl

`ptbl` is a program that will show you the contents of a type table. It can print a table in two modes:

- The value notation for the TBL ASN.1 data structure (see the appendix).
- The ASN.1 text version

Its synopsis is:

```
ptbl [-a] <tbl-file>
```

For example, using `ptbl -a p-rec.tt` to print the PersonnelRecord module used in the last section would yield:

```
P-REC  DEFINITIONS ::=
BEGIN
EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER
Name ::= [APPLICATION 1] IMPLICIT SEQUENCE
{
    givenName IA5String,
    initial IA5String,
    familyName IA5String
}
Date ::= [APPLICATION 3] IMPLICIT IA5String
ChildInformation ::= SET
{
    Name,
    dateOfBirth [0] Date
}
PersonnelRecordSeqOf ::= SEQUENCE OF ChildInformation
PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET
{
    Name,
    title [0] IA5String,
    EmployeeNumber,
    dateOfHire [1] Date,
    nameOfSpouse [2] Name,
    children [3] IMPLICIT PersonnelRecordSeqOf
}
END
--  Definitions for  ASN-USEFUL
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,
    indirect-reference INTEGER,
    data-value-descriptor ObjectDescriptor,
    encoding CHOICE
    {
        single-ASN1-type [0] OCTET STRING,
        octet-aligned [1] IMPLICIT OCTET STRING,
```

```
        arbitrary [2] IMPLICIT BIT STRING
    }
}
END
```

Note that the useful type module is included in the table. As mentioned before, to minimize the size of your tables, put the definitions of the useful types (from `.../asn1specs/asn-useful.asn1`) into your ASN.1 module and do not compile with useful types module (i.e. don't use the -u option). If your module doesn't use any useful types, just don't use the -u option.

The other mode of `ptbl`, value notation for the TBL data structure, produces a lot of output. It may be useful if you are debugging one of the table drivers.

### 12.3.3 Printing Values with pval

The `pval` program will convert BER values into their value notation. Its arguments are a type table file, a module and type name and a list of BER files of that type.

Its synopsis is:

```
pval -T <tt file name> [-m <module name>] -n <type name> <ber value file list>
```

Running `pval` on a PersonnelRecord value produced the following:

```
-- Contents of file "demo-tbls/p-rec.ber"--
-- module = ???, type = PersonnelRecord --

value P-REC.PersonnelRecord ::=
{
  {
    givenName '4a6f686e'H  -- "John" --,
    initial '45'H  -- "E" --,
    familyName '536d697468'H  -- "Smith" --
  },
  title '5468652042696720436865657365'H  -- "The Big Cheese" --,
  99999,
  dateOfHire '3139383230313034'H  -- "19820104" --,
  nameOfSpouse {
    givenName '4d617279'H  -- "Mary" --,
    initial '4c'H  -- "L" --,
    familyName '536d697468'H  -- "Smith" --
  },
  children {
    {
      {
        givenName '4a616d6573'H  -- "James" --,
        initial '52'H  -- "R" --,
        familyName '536d697468'H  -- "Smith" --
      },
```

```
      dateOfBirth '3139353730333130'H  -- "19570310" --
    },
    {
      {
        givenName '4c697361'H  -- "Lisa" --,
        initial '4d'H  -- "M" --,
        familyName '536d697468'H  -- "Smith" --
      },
      dateOfBirth '3139363130363231'H  -- "19610621" --
    }
  }
}

 -- decoded 143 bytes for the above value --
```

## 12.4 Using Tables in Your Own Applications

The best way to get a handle on using tables is to look at the example in `.../tbl-example/`. The general idea is to compile your ASN.1 into a type table (use the snacc `-T` option). If you desire a livable definition of the C data structures for the types in the type table, run `mkchdr` and compile the generated header file with your C code. During runtime, simply load your table file with LoadTblFile (I use the `.tt` suffix naming convention for type table files but it doesn't matter) and then use the TblEncode, TblDecode, TblPrint and TblFree routines with your table. Quite simple. Seriously.

## 12.5 Using GenBufs

The GenBuf generic buffers are really a way of encapsulating other buffer formats. A GenBuf contains a table of pointers to the buffer functions (the standardized ones (see the buffer section) plus "Peek" routines that the table drivers needed). They are defined in `.../c-lib/inc/gen-buf.h`.

GenBufs require functions for the standard buffer routines, macros will not do since you cannot have a pointer to macro.

The benefit of the GenBufs is that since they can support other buffer types, only one set of library routines is needed. (Note that there are 3 libraries in `.../c-lib/` for the backend model and only one for the type table model.

Here is most of `gen-buf.h` to give you an idea of how things work:

```
typedef unsigned char (*BufGetByteFcn) PROTO ((void *b));
typedef unsigned char *(*BufGetSegFcn) PROTO ((void *b,
                                               unsigned long int *lenPtr));
typedef long int (*BufCopyFcn) PROTO ((char *dst, void *b,
                                       unsigned long int len));
typedef void (*BufSkipFcn) PROTO ((void *b, unsigned long int len));
typedef unsigned char (*BufPeekByteFcn) PROTO ((void *b));
```

```
typedef unsigned char *(*BufPeekSegFcn) PROTO ((void *b,
                                                unsigned long int lenPtr));
typedef long int (*BufPeekCopyFcn) PROTO ((char *dst, void *b,
                                    unsigned long int len));
typedef void (*BufPutByteRvsFcn) PROTO ((void *b, unsigned char byte));
typedef void (*BufPutSegRvsFcn) PROTO ((void *b, char *data,
                                     unsigned long int len));
typedef int (*BufReadErrorFcn) PROTO ((void *b));
typedef int (*BufWriteErrorFcn) PROTO ((void *b));

typedef struct GenBuf
{
    BufGetByteFcn    getByte;
    BufGetSegFcn     getSeg;
    BufCopyFcn       copy;
    BufSkipFcn       skip;
    BufPeekByteFcn   peekByte;
    BufPeekSegFcn    peekSeg;
    BufPeekCopyFcn   peekCopy;
    BufPutByteRvsFcn putByteRvs;
    BufPutSegRvsFcn  putSegRvs;
    BufReadErrorFcn  readError;
    BufWriteErrorFcn writeError;
    void             *bufInfo;
    void             *spare; /* hack to save space for ExpBuf ** type */
} GenBuf;

#define GenBufGetByte( b)              ((b)->getByte (b->bufInfo))
#define GenBufGetSeg( b, lenPtr)       ((b)->getSeg (b->bufInfo, lenPtr))
#define GenBufCopy( dst, b, len)       ((b)->copy (dst, b->bufInfo, len))
#define GenBufSkip( b, len)            ((b)->skip (b->bufInfo,len))
#define GenBufPeekByte( b)             ((b)->peekByte (b->bufInfo))
#define GenBufPeekSeg( b, lenPtr)      ((b)->peekSeg (b->bufInfo, lenPtr))
#define GenBufPeekCopy( dst, b, len)   ((b)->peekCopy (dst, b->bufInfo, len))
#define GenBufPutByteRvs( b, byte)     ((b)->putByteRvs (b->bufInfo, byte))
#define GenBufPutSegRvs( b, data, len) ((b)->putSegRvs (b->bufInfo, data, len))
#define GenBufReadError( b)            ((b)->readError (b->bufInfo))
#define GenBufWriteError( b)           ((b)->writeError (b->bufInfo))
```

## 12.6 Type Tables Vs. Metacode

Please refer to section 8.4 on page 119.

# Chapter 13

# Modifying the Compiler

The compiler consists of about 30,000 lines of yacc, lex and C code (another 7,000+ for the runtime library routines). The best way to understand the compiler internals is to understand the module data structure (`.../compiler/core/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:

**lex-asn1.l** add any new keywords

**parse-asn1.y** 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.

In general I have tried to put comments where funky things happen and to use function and variable names that are meaningful. However, things may get ugly in certain places. Thesis writing is harmful to your coding style!

# Chapter 14

# 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 1993 ASN.1
- 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.

1993 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.

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.

Here is a list of the things that I didn't get time to do:

1. Add contexts to nibble memory. It's a pain if you decode a table and then a value—you can't free the value without freeing the table. This one should be relatively easy and only requires modifications to the libraries and the example/user code.

2. Spiff up the table stuff.
   (a) add subtype info (available in big parse tree) to tbl data struct
   (b) add enumerated types name/value info to tbl data struct
   (c) Add ANY DEFINED BY id to type mappings to tbl data struct (available in parse tree if using OBJECT-TYPE macro)
   (d) Support for C++ table drivers.
   (e) I' m sure there's more
3. existing bugs!
   (a) The hash stuff for ANY DEFINED BY mappings will bomb if you go more than 4 levels deep (unlikely, but…). Add linear chaining at the bottom level.
   (b) add the ability to handle CMIP style ANY DEFINED BY ids. Currently snacc gripes about this and stops.

```
Eg ::= SEQUENCE { id Foo, ANY DEFINED BY id }
Foo::= CHOICE { a OBJECT IDENTIFIER, b INTEGER }
```

4. Bigger changes
   (a) punt yacc and lex and use PCCTS (Purdue Compiler Construction…) (better error reporting, easier to deal with (top down), etc.)
   (b) make an ASN.1 '93 version (no more macros!!!)
   (c) add PER.
   (d) make forwards encoders (use only indefinite lengths, though) for C, C++ and tables. Allows simpler buffer writing.

# 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 an 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. See the type table data structure for a format suitable for writing to files.

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 `.../compiler/core/asn1module.h`[1].

```
-- .../asn1specs/asn1module.asn1
--
-- 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
```

[1] The `asn1module.h` that is produced by a current version of Snacc cannot be compiled because its type definitions are in the wrong order. This may be caused by the --snacc cTypeName compiler directives, since one of the affected types is BasicTypeChoiceId, but i'm not really sure. —rj

```
-- 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,
    cxxHdrFileName  MyString,
    cxxSrcFileName  MyString

    cxxname MyString, -- META

    idlFileName MyString, -- IDL
    idlname MyString -- IDL
}


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"
    lineNo INTEGER
```

```
}


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
    lineNo INTEGER
}

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,
    tmpRefCount    INTEGER,
    visited      BOOLEAN,
    definedName  MyString,
    type         Type,
    cTypeDefInfo CTDI,
    cxxTypeDefInfo CxxTDI,
    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,
    cxxTypeRefInfo CxxTRI,
    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,
    isPdu              BOOLEAN,
    isEncDec           BOOLEAN, -- if false, no routines are gen
                                -- and not included in encodings
    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,

    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
}


CxxTDI ::= SEQUENCE
{
    asn1TypeId         INTEGER, --snacc cTypeName:"enum BasicTypeChoiceId"
    className          MyString,
    isPdu              BOOLEAN,
    isEnc              BOOLEAN,
    isPtrForTypeDef    BOOLEAN,
    isPtrForOpt        BOOLEAN,
    isPtrInChoice      BOOLEAN,
    isPtrInSetAndSeq   BOOLEAN,
    isPtrInList        BOOLEAN,
    optTestRoutineName MyString,
    defaultFieldName   MyString -- base for generating field names
}



CxxTRI ::= SEQUENCE
{
    isEnc BOOLEAN,
    className MyString,
    fieldName MyString,
    isPtr BOOLEAN,
    namedElmts CNamedElmts,
    choiceIdSymbol MyString,
    choiceIdValue INTEGER,
    optTestRoutineName MyString
}

IDLTDI ::= SEQUENCE
{
    asn1TypeId         INTEGER, --snacc cTypeName:"enum BasicTypeChoiceId"
    typeName           MyString,
    isPdu              BOOLEAN,
    isEnc              BOOLEAN,
    isPtrForTypeDef    BOOLEAN,
    isPtrForOpt        BOOLEAN,
```

```
    isPtrInChoice      BOOLEAN,
    isPtrInSetAndSeq   BOOLEAN,
    isPtrInList        BOOLEAN,
    optTestRoutineName MyString,
    defaultFieldName   MyString -- base for generating field names
}

IDLTRI ::= SEQUENCE
{
    isEnc BOOLEAN,
    typeName 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
```

# Appendix B

# The Type Table (TBL) Data Structure ASN.1 Definition

The following is the type table data structure that Snacc uses for type table values. Using ASN.1 gives a representation suitable for saving tables to files or sending them over a network to reconfigure a device (e.g. SNMP mib).

This file is actually compiled by Snacc to compile itself. For bootstrapping purposes, an initial version is included in the distribution.

```
-- .../asn1specs/tbl.asn1
--
--  TBL types describe ASN.1 data structures.
--  These can be used in generic, interpretive encoders/decoders.
--  Interpretive decoders are typically slower, but don't eat memory
--  with type-specific encoding and decoding code.
--  The tbl types can also be sent over the network
--  and allow dynamic re-configuration of encoders/decoders.
--
--  This definition is fairly small so it should be reasonable easy
--  to understand.  To learn more about semantics of this data
--  struct, look in snacc/tbl-tools/print-tbl/pasn1.c.
--
--  Copyright Mike Sample and UBC, 1992, 1993
--

TBL DEFINITIONS ::=
BEGIN


-- imports nothing
-- exports nothing

TBL ::= --snacc isPdu:"TRUE" -- SEQUENCE
{
   totalNumModules  INTEGER,  -- these totals can help allocation
```

```
   totalNumTypeDefs INTEGER,  -- when decoding (ie use arrays)
   totalNumTypes    INTEGER,
   totalNumTags     INTEGER,
   totalNumStrings  INTEGER,
   totalLenStrings  INTEGER,
   modules SEQUENCE OF TBLModule
}

TBLModule ::= SEQUENCE
{
    name     [0] IMPLICIT PrintableString,
    id       [1] IMPLICIT OBJECT IDENTIFIER OPTIONAL,
    isUseful [2] IMPLICIT BOOLEAN, -- true if useful types module
    typeDefs [3] IMPLICIT SEQUENCE OF TBLTypeDef
}

TBLTypeDef ::= SEQUENCE
{
   typeDefId TBLTypeDefId,
   typeName  PrintableString OPTIONAL, -- I have forgotten why this is opt!
   type      TBLType
}

TBLType ::= SEQUENCE
{
   typeId    [0] IMPLICIT TBLTypeId,
   optional  [1] IMPLICIT BOOLEAN,
   tagList   [2] IMPLICIT SEQUENCE OF TBLTag OPTIONAL,
   content   [3] TBLTypeContent,
   fieldName [4] IMPLICIT PrintableString OPTIONAL
}

TBLTypeContent ::= CHOICE
{
   primType [0] IMPLICIT NULL,
   elmts    [1] IMPLICIT SEQUENCE OF TBLType,
   typeRef  [2] IMPLICIT TBLTypeRef
}

TBLTypeRef ::= SEQUENCE
{
    typeDef TBLTypeDefId,
    implicit BOOLEAN
}

TBLTypeId ::= ENUMERATED
{
    tbl-boolean (0),
    tbl-integer (1),
    tbl-bitstring (2),
    tbl-octetstring (3),
    tbl-null (4),
    tbl-oid (5),
    tbl-real (6),
```

```
    tbl-enumerated (7),
    tbl-sequence (8),
    tbl-set (9),
    tbl-sequenceof (10),
    tbl-setof (11),
    tbl-choice (12),
    tbl-typeref (13)
}

TBLTypeDefId ::= INTEGER

TBLTag ::= SEQUENCE
{
    tclass TBLTagClass,
    code INTEGER (0..MAX)
}

TBLTagClass ::=  ENUMERATED { universal (0), application (1),
                              context (2), private (3)}

END
```

# Appendix C

# ASN.1 Files for the Editor Example

The files can be found in `.../tcl-example/`.

```
-- file: edex0.asn1
--
-- SnaccEd example, simple types module

EdEx-Simple DEFINITIONS ::=
BEGIN

RainbowColor ::= INTEGER
{
   red(0), orange(1), yellow(2), green(3), blue(4), indigo(5), violet(6)
}

DayOfTheWeek ::= ENUMERATED
{
   sunday(0), monday(1), tuesday(2), wednesday(3), thursday(4), friday(5), saturday(6)
}

Hand ::= BIT STRING
{
   thumb(0), forefinger(1), middle-finger(2), ring-finger(3), little-finger(4)
}

victory Hand ::= { forefinger, middle-finger }

END


-- file: edex1.asn1
--
-- SnaccEd example, structured types module
```

```
EdEx-Structured DEFINITIONS ::=
BEGIN

IMPORTS RainbowColor, DayOfTheWeek, Hand FROM EdEx-Simple;

RGBColor ::= SEQUENCE
{
   red INTEGER,
   green INTEGER,
   blue INTEGER
}

Coordinate ::= CHOICE
{
   cartesian [0] SEQUENCE { x REAL, y REAL },
   polar [1] SEQUENCE { angle REAL, distance REAL }
}

File ::= SET
{
   name [0] PrintableString,
   contents [1] OCTET STRING,
   checksum [2] INTEGER OPTIONAL,
   read-only [3] BOOLEAN DEFAULT FALSE
}

Directory ::= SET
{
   name PrintableString,
   files SET OF File
}

Simple ::= SET
{
   null [0] NULL,
   bool [1] BOOLEAN,
   day [2] DayOfTheWeek,
   int [3] INTEGER,
   color [4] RainbowColor,
   real [5] REAL,
   bits [6] Hand,
   str [7] OCTET STRING,
   optstr [8] OCTET STRING OPTIONAL
}

Structured ::= SET
{
   coord [0] Coordinate,
   color [1] CHOICE { rainbow RainbowColor, rgb RGBColor }
}

Various ::= SET
{
   simple [0] Simple,
```

```
   struct [1] Structured,
   recursion [2] Various OPTIONAL
}

END
```

# Appendix D

# Coding Tricks For Readability

One of our project partners needed some additional function arguments and so they duplicated the function declarator and put a preprocessor switch around it. The metacode and the Tcl interface added some additional compilation conditions. Since the Tcl interface is only useful on top of the metacode, there are six different combinations instead of eight. Even these six combinations would have made the code very ugly. Please decide for yourself, here is an example with an example function with only up to four arguments, PrintCxxCode() has up to 17 arguments.

```
static void
#if A
FunctionName PARAMS ((a, b),
        TypeA a _AND_
        TypeB b)
#if B
FunctionName PARAMS ((a, b, c),
        TypeA a _AND_
        TypeB b _AND_
        TypeC c)
#if C
FunctionName PARAMS ((a, b, c, d),
        TypeA a _AND_
        TypeB b _AND_
        TypeC c _AND_
        TypeD d)
#endif
#endif
#else
FunctionName PARAMS ((b),
        TypeB b)
#if B
FunctionName PARAMS ((b, c),
        TypeB b _AND_
        TypeC c)
#if C
FunctionName PARAMS ((b, c, d),
```

```
        TypeB b _AND_
        TypeC c _AND_
        TypeD d)
#endif
#endif
#endif
```

Here is the code after introduction of my shorthand:

```
static void
FunctionName PARAMS ((if_A (a COMMA) b if_C (COMMA c) if_D (COMMA d)),
        if_A (TypeA a _AND_)
        TypeB b
        if_C (_AND_ TypeC c)
        if_D (_AND_ TypeD d))
```

The tricks are very simple. One is the if_… macro

```
#if A
#define if_A( code)                     code
#else
#define if_A( code) #endif
```

that lets us get rid of at least four lines of code for every invocation, and the other trick is the COMMA macro

```
#define COMMA                           ,
```

that makes the arguments to the if_… macros look like a single argument. Without this trick,

The other trick, of course is the if_… macro itself. The if_… macros have to expand into code without brackets, for example if_A (a COMMA b) exands into a, b. The COMMA is not my invention, snacc's _AND_ macro is exactly the same. Both _AND_ and COMMA serve the purpose of being a comma (“,”) as the final result (well, only for ANSI C, for K&R C, the _AND_ becomes a semicolon), but without being an argument separator to the C preprocessor. The PROTO macro that was already present in snacc 1.1 gets a single argument as well, but by means of additional parenthesis, inside which commas can savely be used. It expands into code with brackets around it: PROTO ((int a, char *b)) expands into (int a, char *b). The first argument to the PARAM macro is bracketed list as well, and for the arguments purpose, to be a function argument list, this is fine.

To have both an ANSI C and a K&R C version, without PROTO, PARAMS and supporting macros, twelve conditional code compilations would have to be written out instead of one! And what a tedious job to maintain all twelve versions!

# Appendix E

# Makefiles

Some of Snacc's makefiles look rather sophisticated. This section explains some of the tricks.

## E.1 CVS, Dependencies and Make's Include Statement

The makefiles take advantage of the file inclusion feature. Since this has already been supported by UNIX System III[1] make (somewhen around 1980), I consider it to be pretty portable. If your make is crippled, either use a newer one (e.g. GNU make), or as a last resort, remove (better: comment out) the include statements and call make with the additional arguments `-f .../makehead -f makefile -f dependencies -f .../maketail`.

Snacc's configuration script generates the file `makehead` which gets included by all makefiles. It contains a lot of definitions used by make.

The dependencies have been moved out of each makefile into a separate file called `dependencies` that is not under cvs control—otherwise, the makefiles would inflate the repository unnecessarily. The makefiles have an include statement for their dependencies file. GNU make automatically makes the dependencies if the file does not exist, but other versions of `make` simply give up. In that case, an initial (empty) file has to be generated. Snacc's top level makefile does this for you if you call `make depend`.

A third file that is included by almost every makefile is `.../maketail`. It holds the rules that are common to all makefiles where C/C++ code is compiled.

[1] yes, System III, not System V R3

## E.2 Circular Dependencies

In a normal makefile rule, a file depends upon other files. If any of a file's dependencies is newer, the file is remade. This goes well as long as the dependency graph is non-circular, but snacc is compiled from some files it has generated itself. This recursion can lead to one of two results: in the worse case, `make` builds the compiler because its source files are newer, builds the source files because the compiler is newer, builds the compiler because some source files are newer, and so on ad infinitum... Even if this endless recursion does not happen, one or two of the above steps will be made every time `make` is called. To avoid this waste of time, one lets the compiler generate a new source file, but when the new and the old version are identical, the old file is kept and `make` sees that the compiler is up-to-date, and the recursion is terminated. Of course, if the source file's contents did change, it is replaced with the new version.

This is a simplified example of a normal makefile:

```
snacc:          tbl.h
                compile snacc

tbl.h:          snacc tbl.asn1
                ./snacc ...  tbl.asn1
```

Most `make` versions will complain and print a warning about this `infinite loop' or `circular dependency'. The first approach towards a solution could be:

```
snacc:          tbl.h
                compile snacc

tbl.h:          snacc tbl.asn1
                mv tbl.h tbl.h.prev
                ./snacc ...  tbl.asn1
                if cmp tbl.h.prev tbl.h; then\
                  echo "tbl.h hasn't changed";\
                  mv tbl.h.prev tbl.h;\
                else\
                  $(RM) tbl.h.prev;\
                fi
```

The effect is that you keep snacc from being remade if the contents of tbl.h did not change, but the two steps to create tbl.h and to test whether it is different from tbl.h.prev will be made every time snacc or tbl.asn1 are newer than tbl.h, which they most often will be since few of the changes to snacc will affect tbl.h's contents. And `make` will still complain about the recursion. To solve all this, another file, a stamp file is introduced. It separates the file's contents from its modification time:

```
snacc:          tbl.h
                compile snacc
```

```
stamp-tbl:      snacc tbl.asn1
                mv tbl.h tbl.h.prev
                ./snacc ...  tbl.asn1
                if cmp tbl.h.prev tbl.h; then\
                  echo "tbl.h hasn't changed";\
                  mv tbl.h.prev tbl.h;\
                else\
                  $(RM) tbl.h.prev;\
                fi
                date > $@

tbl.h:          stamp-tbl
                @true
```

The dummy command in the rule for tbl.h is necessary, since otherwise, despite stamp-tbl commands having modified tbl.h, many versions of make think that tbl.h has not been modified.

If you want `tbl.h` to be remade (e.g. you have changed an option to snacc), you must delete `stamp-tbl`—`tbl.h` may (and should) be left in place.

The rules in `.../compiler/makefile`, `.../c-lib/makefile` and `.../c++-lib/makefile` are further complicated by the fact that

1. snacc prints the current time into the file which the comparison must take into account
2. if snacc has not been built it cannot be used to generate its source files—a bootstrapping version of snacc's source files has got to be supplied.

## E.3 Compiling Different Libraries From One Set Of Source Files

The different libraries in `.../c-lib/` and `.../c++-lib/` get made by means of recursive calls to make with different macro settings. This keeps the makefiles short as it avoids a lot of duplication of file lists and rules which would be a hassle to maintain. The different libraries get compiled from the same set of source files, the code to be compiled is determined through `cpp` (C preprocessor) macro switches.

## E.4 Configuration, Optional Code and Makefiles

The `.../configure` script looks for Tcl/Tk. If they are absent, there is no use in trying to compile Snacc's Tcl interface. For makefiles to detect whether the Tcl interface should be compiled or not, there is a file `.../tcl-p.c` that, after being compiled into `tcl-p`, exits with 0 (the shells' `true' value) if Tcl/Tk is present and the

user has not disabled this option by setting NO_TCL in .../policy.h to 1. tcl-p gets made automatically.

# Bibliography


[1] CCITT. *Data Communications Networks Open systems Interconnection (OSI) Model and Notation, Service Defintion*, chapter Recommendation X.208, Specification of Abstract Syntax Notation One (ASN.1), pages 57–130. Number Fascicle VIII.4 in Blue Book. 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*, chapter Recommendation X.209, Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), pages 130–151. Number Fascicle VIII.4 in Blue Book. 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] ISO. Information technology—open systems interconnection—abstract syntax notation one (asn.1).

[5] ISO. Working paper for draft proposed international standard for information systems—programming language c++, 28 April 1995.

[6] Gerald Neufeld and Son Vuong. An overview of asn.1. *IEEE Networks and ISDN Systems*, 23(5):393–415, Feb 1992.

[7] Gerald Neufeld and Yeuli Yang. An asn.1 to c compiler. *IEEE Transactions on Software Engineering*, 16(10):1209–1220, Oct 1990.

[8] OMG. The common object request broker: Architecture and specification. Technical report, OMG, 1993.

[9] John K. Ousterhout. *Tcl and the TK Toolkit*. Addison-Wesley Publishing Company, 1994. ISBN 0-201-63337-X.

[10] M. Rose and K. McCloghrie. Structure and identification of management information for tcp/ip-based internets (rfc 1155). Network Information Center, SRI International, May 1990.

[11] Marshall T. Rose. *ISODE, The ISO Development Environment: User Manual*. Wollongong Group, 1129 San Antonio Rd. Palo Alto, California, USA, February 1990.

[12] Michael Sample. How fast can asn.1 encoding rules go? Master's thesis, University of British Columbia, Vancouver, B.C. Canada V6T 1Z2, April 1993.

[13] Michael Sample and Gerald Neufeld. Implementing efficient encoders and decoders for network data representations. *IEEE INFOCOM '93 Proceedings*, 3:1144–1153, Mar 1993.

[14] Douglas Steedman. *ASN.1, The Tutorial and Reference*. Technology Appraisals Ltd., 1990. ISBN 1 871802 06 7.

[15] Bjarne Stroustrup. *The C++ Programming Language, 2nd Edition*. Addison-Wesley Publishing Co., 1991. ISBN 0201539926.