

# Design and Implementation of a Generic ASN.1 Editor

Robert Joop

Diploma Thesis

Technische Universität Berlin  
Fachbereich 13 (Informatik)  
Institut für Software und Theoretische Informatik  
Offene Kommunikationssysteme (OKS)  
Prof. Dr. Radu Popescu-Zeletin  
Franklinstraße 28/29  
D-10587 Berlin

September 1995



# Zusammenfassung

## **Design and Implementation of a Generic ASN.1 Editor**

(Entwurf und Verwirklichung eines generischen ASN.1-Editors)

Die Struktur vieler in der Kommunikationstechnik ausgetauschter Daten wird in ASN.1 (Abstract Syntax Notation One) beschrieben. In dieser formalen, maschinell verarbeitbaren Sprache beschriebene Typdefinitionen werden von ASN.1-Compilern in äquivalente Datenstrukturen einer Zielsprache wie z.B. C++ umgeformt. Ebenfalls von diesen Compilern erzeugte Funktionen konvertieren ASN.1-Daten zwischen ihrer rechner-spezifischen internen Darstellung und ihrem rechnerunabhängigen Austauschformat, das meist durch BER (Basic Encoding Rules) definiert ist.

Das binäre BER-Format ist nicht menschenlesbar. Problematisch wird dies, wenn bei Programmierfehlern die zwischen kommunizierenden Programmen ausgetauschten Daten untersucht werden müssen. Ferner wäre es nützlich, wenn Daten für Testzwecke erzeugt oder modifiziert werden könnten, ohne dafür extra spezielle Programme schreiben zu müssen.

Diese Diplomarbeit beschreibt den Entwurf und die Entwicklung eines Editors für diese Zwecke. Seine grafischen Elemente erlauben sowohl die Darstellung der Daten entsprechend ihrer Struktur und ihres Typs als auch deren Veränderung. Der Darstellungsausschnitt kann gezielt gewählt werden; umfangreiche Daten werden so handhabbar.

Der Editor basiert auf zwei Hauptkomponenten: Snacc, einem ASN.1-Compiler, und einer grafischen Benutzeroberfläche.

Mit Snacc können C++-Quelldateien mit Typ- und Funktionsdefinitionen erzeugt werden. Ein C++-Compiler übersetzt diese Dateien, und die Namen von Datentypen und deren Komponenten werden zu Speicheradressen. Um auch noch im übersetzten Programm auf die ursprünglichen Namen zugreifen zu können, mußte der ASN.1-Compiler so erweitert werden, daß er die erzeugten C++-Klassen mit Information über ihre eigene Struktur ausstattet, mit „Metacode“. Mit Hilfe virtueller Funktionen konnte eine speicher- und recheneffiziente Implementierung verwirklicht werden.

Die grafische Benutzeroberfläche wurde mit Hilfe von Tk, einer kostenlos erhältlichen Implementierung des Motif-Look-and-Feel, geschaffen. Hinzugenommen wurde ein als Freeware erhältliches Tree-Widget, mit dessen Hilfe die hierarchische Struktur der ASN.1-Daten dargestellt wird.

Sämtliche Komponenten, die Snacc-Funktionen, Tk und das Tree-Widget, wurden in die Erweiterungssprache Tcl eingebettet.

Der generische Editor besteht aus einer Anzahl von Tcl-Skripten. Sie steuern den Aufbau der grafischen Oberfläche, lesen und schreiben die ASN.1-Daten und erlauben deren Veränderung.

Soweit bekannt ist, handelt es sich hier um den ersten freeware ASN.1-Editor; er kann mit Hilfe kostenlos erhältlicher Tools gebaut werden.



# Abstract

## Design and Implementation of a Generic ASN.1 Editor

In the field of data communication the structure of the data exchanged is often described in ASN.1 (Abstract Syntax Notation One). ASN.1 is a formal, machine parseable language. ASN.1 compilers turn ASN.1 type definitions into equivalent data structures in a target language, C++ for example. Functions generated by the same compiler convert ASN.1 data between their machine dependent in-core representation and their external architecture independent format, that most often is defined through BER (Basic Encoding Rules).

BER is a binary format that is not human readable. This becomes a problem when in case of programming errors the data that is exchanged between communicating programs has to be examined. Moreover it would be useful if, for testing purposes, data could be generated or modified without having to write special programs.

This diploma thesis describes the design and implementation of an editor for these purposes. This editor displays ASN.1 data and allows their modification using graphical elements according to the data's structure and type. The user can select which and how much of the data is to be displayed; voluminous data becomes manageable.

The editor is built based on two main components: Snacc, an ASN.1 compiler, and a graphical user interface.

Snacc can be used to generate C++ source files with type and function definitions. A C++ compiler translates these files, and the names of data types and their components become memory addresses. For the original names to be accessible in the compiled program even after compilation, the ASN.1 compiler had to be augmented: the generated data structures need information about their own structure, some "metacode". Using virtual functions a space and time efficient implementation could be realized.

The graphical user interface was created with the help of Tk, an unencumbered implementation of the Motif look and feel. A freeware tree widget was added that helps to display the hierarchical structure of the ASN.1 data.

All the components, the Snacc functions, Tk and the tree widget were embedded in Tcl, an extension language.

The generic editor consists of a number of Tcl scripts. They control the user interface, they read and write the ASN.1 data, and they allow the data to be modified.

To the best of my knowledge this is the first ASN.1 editor that is free and that can be built using tools that are freeware or unencumbered software.



# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>                               | <b>1</b>  |
| 1.1      | Typographic Conventions . . . . .                 | 6         |
| <b>2</b> | <b>An Abridge Introduction into ASN.1 and BER</b> | <b>7</b>  |
| 2.1      | ASN.1 Types . . . . .                             | 9         |
| 2.1.1    | Simple Types . . . . .                            | 9         |
| 2.1.1.1  | NULL . . . . .                                    | 9         |
| 2.1.1.2  | BOOLEAN . . . . .                                 | 10        |
| 2.1.1.3  | INTEGER . . . . .                                 | 10        |
| 2.1.1.4  | ENUMERATED . . . . .                              | 10        |
| 2.1.1.5  | REAL . . . . .                                    | 11        |
| 2.1.1.6  | BIT STRING . . . . .                              | 11        |
| 2.1.1.7  | OCTET STRING . . . . .                            | 11        |
| 2.1.1.8  | Character String . . . . .                        | 12        |
| 2.1.1.9  | OBJECT IDENTIFIER . . . . .                       | 12        |
| 2.1.2    | Structured Types . . . . .                        | 13        |
| 2.1.2.1  | SET and SEQUENCE . . . . .                        | 13        |
| 2.1.2.2  | SET OF and SEQUENCE OF . . . . .                  | 14        |
| 2.1.2.3  | CHOICE . . . . .                                  | 14        |
| 2.1.2.4  | ANY and ANY DEFINED BY . . . . .                  | 14        |
| 2.1.3    | Useful Types . . . . .                            | 15        |
| <b>3</b> | <b>Snacc</b>                                      | <b>17</b> |
| 3.1      | General Overview . . . . .                        | 17        |
| 3.1.1    | The Snacc Compiler . . . . .                      | 18        |
| 3.1.2    | The Snacc Libraries . . . . .                     | 18        |
| 3.2      | Evolution from Version 1.1 to 1.2rj . . . . .     | 18        |
| 3.3      | ASN.1 to C++ Naming Conventions . . . . .         | 20        |

|          |  |           |
|----------|--|-----------|
| <b>4</b> | <b>The Metacode</b>                            | <b>23</b> |
| 4.1      | Implementation . . . . .                       | 24        |
| 4.1.1    | Named Values . . . . .                         | 32        |
| 4.1.2    | Types with Members . . . . .                   | 33        |
| 4.1.3    | SET OF and SEQUENCE OF . . . . .               | 35        |
| 4.1.4    | Aliases . . . . .                              | 37        |
| 4.1.5    | ANY (DEFINED BY) . . . . .                     | 38        |
| 4.1.6    | Modules . . . . .                              | 38        |
| 4.2      | Efficiency . . . . .                           | 39        |
| 4.2.1    | Normal Operation . . . . .                     | 39        |
| 4.2.2    | Metacode . . . . .                             | 39        |
| 4.3      | Metacode vs. Type Tables . . . . .             | 40        |
| <b>5</b> | <b>Tcl and Tk</b>                              | <b>43</b> |
| 5.1      | Problems and Pitfalls . . . . .                | 43        |
| 5.1.1    | The Input Parser . . . . .                     | 43        |
| 5.1.1.1  | Quoting . . . . .                              | 45        |
| 5.1.2    | Non-orthogonal commands . . . . .              | 47        |
| 5.1.3    | The Trace Mechanism . . . . .                  | 49        |
| 5.1.4    | Name Space . . . . .                           | 51        |
| 5.2      | Conclusion . . . . .                           | 52        |
| <b>6</b> | <b>Tcl Interface</b>                           | <b>53</b> |
| 6.1      | The <code>snacc</code> Tcl command . . . . .   | 54        |
| 6.1.1    | File Commands . . . . .                        | 54        |
| 6.1.2    | Generic Information Retrieval . . . . .        | 56        |
| 6.1.3    | Operations on Contents and Structure . . . . . | 57        |
| 6.2      | An Example Session . . . . .                   | 59        |
| 6.3      | Implementation . . . . .                       | 61        |
| 6.3.1    | Lower Layer . . . . .                          | 61        |
| 6.3.1.1  | Generic Information Retrieval . . . . .        | 62        |
| 6.3.1.2  | Operations on Contents and Structure . . . . . | 62        |
| 6.3.2    | Upper Layer . . . . .                          | 63        |
| 6.4      | Setup for the Tcl Code Generator . . . . .     | 64        |
| 6.5      | Deficiencies . . . . .                         | 64        |

|  |           |
|--|-----------|
| <i>CONTENTS</i>  | ix        |
| <b>7 SnaccEd, the Snacc Editor</b>                                       | <b>65</b> |
| 7.1 Manipulating the Display . . . . .                                   | 67        |
| 7.2 The Content Window . . . . .   | 68        |
| 7.3 Building Your Own Editor . . . . .                                   | 71        |
| 7.4 Implementation . . . . .   | 72        |
| 7.4.1 Procedures . . . . .   | 73        |
| 7.4.2 Data Structures . . . . .  | 74        |
| 7.5 Future Work . . . . .  | 76        |
| <b>A Coding Tricks For Readability</b>                                   | <b>79</b> |
| <b>B Makefiles</b>   | <b>81</b> |
| B.1 CVS, Dependencies and Make's Include Statement . . . . .             | 81        |
| B.2 Circular Dependencies . . . . .                                      | 81        |
| B.3 Compiling Different Libraries from One Set of Source Files . . . . . | 83        |
| B.4 Configuration, Optional Code and Makefiles . . . . .                 | 83        |
| <b>C ASN.1 Files for the Editor Example</b>                              | <b>85</b> |



# List of Figures

|     |   |    |
|-----|---|----|
| 1.1 | Building Scheme: Snaccwish and Application . . . . .                  | 3  |
| 1.2 | SnaccEd: An Example Screen Shot (1) . . . . .                         | 4  |
| 7.1 | SnaccEd: An Example Screen Shot (2) . . . . .                         | 65 |
| 7.2 | The File And Content Type Selection Box . . . . .                     | 67 |
| 7.3 | Content editors for ASN.1 Simple Types . . . . .                      | 69 |
| 7.4 | Content Editors for ASN.1 Structured Types . . . . .                  | 70 |
| 7.5 | Popup for Import/Export of OCTET STRING Contents . . . . .            | 70 |
| 7.6 | Popup for Action Selection for SET OF and SEQUENCE OF Types . . . . . | 70 |
| 7.7 | Data in a Recursive ASN.1 Type . . . . .                              | 76 |



# Chapter 1

## Introduction

This thesis is about a generic editor for ASN.1 data files.

Abstract Syntax Notation One (ASN.1) provides a platform independent language to describe hierarchical data structures, especially for data that is to be transmitted in communication protocols. ASN.1 does not imply any implementation or data format. It is a formal, machine parseable language and compilers have been built that read the abstract type descriptions and transform them into concrete data structures and/or functions that transform ASN.1 data between their machine specific in-core representation and an architecture independent external representation. Chapter 2 gives a brief introduction to ASN.1.

The external representation that is most often used is BER, the Basic Encoding Rules. BER is binary format, it is not human readable.

If protocols were described in terms of bits and bytes, programmers would have to concern themselves with, for example, their hardware's word length, byte order and floating point representation and how their compilers pack data structures. They would have to write software that transforms data between their machine's representation and the format that is defined in the protocol definition.

Given an ASN.1 description and an ASN.1 compiler that adheres to the Basic Encoding Rules, they can save a great deal of work and start at a higher level of abstraction.

Chapter 3 describes Snacc, an ASN.1 compiler. Snacc has been developed by Michael Sample at the University of British Columbia (Canada). In 1993 he released it to the public as freeware, and it became quite popular since then. Snacc is no longer maintained by its original author.

Snacc is able to turn ASN.1 descriptions into C or C++ code.

Snacc has been picked by the GLASS project as its ASN.1 compiler of choice. GLASS is a joint scientific project of Digital CEC (Karlsruhe), GMD FOKUS (Berlin), Grundig Multimedia Solutions (Nürnberg), IBM ENC (Heidelberg) and PRZ der Technischen Universität Berlin; it is funded by DeTeBerkom (Berlin). GLASS, short for "GLobally Accessible Services", is a distributed multimedia system. Within this project, Snacc is used to generate C++ code. IBM ENC made a couple of enhancements to Snacc's C++ code generator: destructors got added to the generated C++ classes and huge inline functions were turned into normal functions to reduce the size of the generated object code.

During the writing of this thesis, Snacc's code was reorganized, duplicate configuration files and libraries got merged, the configuration was simplified and the C++ code generator got completed. An additional code generator for CORBA IDL is currently being developed.

The resulting Snacc version has been released to the public and has met with positive response. It is available as `ftp://ftp.fokus.gmd.de/pub/freeware/snacc/snacc-1.2rj.patchlevel.tar.gz`.

Out of the GLASS project rose the desire for a simple tool for browsing of BER encoded PDUs (Protocol Data Units). The binary BER format is problematic when in case of programming errors the data that is exchanged between communicating programs has to be examined. Moreover, it would be useful if, for testing purposes, data could be generated or modified without having to write special programs: the tool should be an editor.

The editor should not be designed for a certain set of ASN.1 files; it should be generic so that ASN.1 data of arbitrary type can be displayed and modified in a consistent user interface. The user interface should be a graphical, not a text based one.

Snacc is the underlying utility for this editor; the Compiler has been augmented so that the ASN.1 data and their descriptions become accessible to generic programs. Since this task was best realized using C++' virtual functions, only Snacc's C++ part got extended.

Snacc can be used to generate C++ source files with type and function definitions. A C++ compiler translates these files, and the names of data types and their components become memory addresses. For the original names to be accessible in the compiled program even after compilation, the Snacc generated data structures had to be augmented and some access functions had to be added. This additional information describes the C++ classes themselves, and therefore this code was termed "metacode". In the ASN.1 compiler, both the structural information and the names in string form are present—they are necessary to generate the C++ source files. The metacode is described in Chapter 4.

The graphical user interface should be built using publically available software. Tcl/Tk was chosen as it is unencumbered software that implements an appealing widget set, the Motif look and feel. Tcl, short for Tool Command Language, is an embeddable extension language. Tk is the best known Tcl extension, it implements the widget set. Chapter 5, Tcl and Tk, is less of an introduction, but more of a warning: Tk is a valuable toolkit, but think about whether you need Tcl to use it.

Since the user interface is built using Tcl/Tk, for a seamless integration the metacode is best made accessible as a Tcl command. Snacc's Tcl interface is a Tcl extension that provides this new command. It allows ASN.1 data to be examined and manipulated. The Tcl interface is discussed in Chapter 6.

Chapter 7 describes the ASN.1 editor. The editor consists of two parts, an executable and a set of Tcl scripts.

The executable is combined from the Tcl core and three extensions: the Snacc routines that are specific to the ASN.1 description, the Tk widget set and the tree widget. The editor's executable will be referred to as `snaccwish`, because the standard executable that is combined from linking the Tcl core with the Tk widget set has got the name `wish` for "windowing shell".

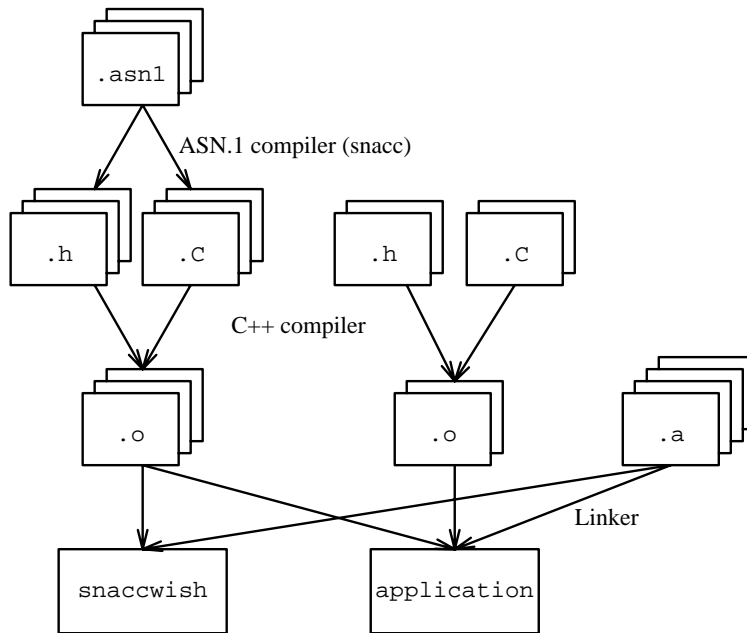


Figure 1.1: Building Scheme: Snaccwish and Application

Figure 1.1 is a simplified scheme of both the snaccwish's and a normal application's building process. Both programs can be compiled from the same set of Snacc object files, the metacode and the Tcl interface code do not prevent the Snacc code's normal operation. The snaccwish is compiled from the Snacc generated files and a few libraries only. The application will be linked with some additional object files that make up the application's specific functionality. The application usually will use neither metacode nor Tcl interface routines, but as long as the application is under development, the time for an additional compilation step where the metacode has been disabled can be saved.

The Tcl scripts implement the generic editor. They build and control the user interface, they read and write the ASN.1 data, and they allow the data to be modified. The Tk and tree widgets are combined to a graphical user interface where the ASN.1 data is shown using graphical elements that elucidate the data's structure and types. Many changes to the data can be made by clicking at graphical elements. The user can select which and how much of the data is to be displayed; voluminous data becomes manageable.

ASN.1 data always has got a tree structure. The type definitions may be recursive using CHOICE types or OPTIONAL components, and a PDU may contain instances of a type that contains other instances of the same type, but as ASN.1 has not got any pointers, cycles are impossible. The example description's Misc type is recursive, and the example file `misc.ber` that is shown in Figure 1.2 contains two nested values of this type: they are labeled `misc.ber` and `recursion`.



```

Direction ::= ENUMERATED { left(0), down(1), up(2), right(3) }

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

Misc ::= SET
{
  name PrintableString,
  recursion Misc OPTIONAL,
  scan-line SEQUENCE OF Color,
  flip-flop BOOLEAN,
  where Direction OPTIONAL,
  calculator Hand
}

END

```

The editor uses different graphical elements to display the various ASN.1 types. The example shows some of them:

- a BOOLEAN value is displayed as a toggle button, like `flip_flop`.
- INTEGER types have an area where the numeric value can be seen and changed. The example's list elements at the right hand side are INTEGERS with named values. They have got an additional radio button area where the object's value can be selected by clicking at a button.
- ENUMERATED types are displayed in an arrangement of radio buttons. The numeric value is not shown, `where` is an example.
- In BIT STRINGS, `calculator` is an example, individual bits can be toggled using the buttons, and the binary representation can be manipulated in an entry area.
- Character Strings like `name` can be edited in a text widget.
- SET types are displayed as a number of toggle buttons. `Misc` is such a type. OPTIONAL components can be allocated or deleted by clicking at the button. Mandatory components cannot be deleted—their buttons are greyed out and do not respond to button clicks.
- SEQUENCE OF types are displayed in a list box widget. Individual list elements can be selected for display. In the `scan_line` example, four list elements are selected for display and two of them have got an opened content editor.

The window below the node name that shows the node's contents is not always shown, it has to be opened by clicking at the node name and it can be closed by the same action.

At every node in the data hierarchy the display can be clipped on both sides so that the node can be displayed as the root of its subtree and/or as a leaf.

This ASN.1 editor is meant as a debugging tool for programmers, not as an end user application. For end users, the details of a data communication protocol are irrelevant. Showing every tiny detail will usually provoke confusion.

## 1.1 Typographic Conventions

Throughout this thesis, the following typographic conventions are applied:

The typewriter face *Courier* is used for:

- program names and their options and arguments, pathnames and filenames
- URLs (Universal Resource Locators)
- examples of interpreted code: Bourne shell scripts, Tcl shell scripts, Perl scripts, makefiles.

The sans serif typeface Helvetica is used for:

- Code that is to be compiled: ASN.1 descriptions and C++ code.

*Times-Italic* and *Courier-Oblique* are used for:

- non-literal text: the written text can be replaced with any text that fits the given description. For example, *snacc files* may have a legal replacement `snacc edex0.asn1 edex1.asn1`.

## Chapter 2

# An Abridge Introduction into ASN.1 and BER

This chapter gives a brief introduction to ASN.1, and how it is mapped into C++ by Snacc. Snacc is an ASN.1 to C/C++ compiler, it will be introduced in the following chapter. Snacc's C mapping is omitted for brevity, its description can be found in the Snacc user documentation [13]. ASN.1, or “Abstract Syntax Notation One”, serves the purpose of describing system independent data structures. ASN.1 describes how some piece of information is organized, but it does not describe how it is to be represented. The data representation is the subject of the “Basic Encoding Rules”, or BER for short.

BER defines a system independent representation for values. Since the application designer has not to be familiar with BER details, I will not go into explaining them, just mention that

- BER is a binary encoding and therefore the data is not human readable.
- it contains not just the data itself but each value is preceeded by a type identifier. This makes it possible to detect coding errors such as giving data that belongs to some ASN.1 description to a different ASN.1 description's decoding routine.

While in the mean time, additional encodings have been developed, BER is the only encoding supported by Snacc.

This chapter presents only a brief introduction to ASN.1 and how Snacc deals with it— for a more thorough introduction and a tutorial to both ASN.1 and BER please refer to Douglas Steedman's ASN.1 book [14]. For an online document, have a look at “A Layman's Guide to a Subset of ASN.1, BER, and DER” [8], which can be found under the URL <ftp://ftp.uni-erlangen.de/pub/doc/ISO/ASN.1.ps.Z>. This site has got a lot of standards related stuff (freely available documents only; unfortunately, ISO standards have to be paid for).

ASN.1 is defined in international standards, CCITT Recommendation X.208 [3] and ISO 8824 [5]. BER is defined in CCITT Recommendation X.209 [4] and ISO 8825 [6].

ASN.1 is most often used to define the data structures in CCITT and ISO standards.

- MHEG, for coding of multimedia and hypermedia objects, the underlying standard of the GLASS project (this ISO standard is not yet finished, currently it is a draft international standard (DIS))
- MHS , Message Handling Systems, or better known as X.400 Mail (CCITT Series of Recommendations)
- ODA, Open Document Architecture, for the exchange of documents (ISO 8613)
- PKCS, Public Key Cryptography Standard (RSA Data Security, Inc.)
- SNMP, Simple Network Management Protocol (this is an Internet standard, defined in RFC 1157)
- X.500 Series of Recommendations: Directory, Authentication

Just like C programs are usually compiled from a number of C source files, ASN.1 descriptions may be split into several files, called modules. Every module contains zero or more type definitions and may contain value definitions for the types defined. Types and values may be private to a module or they can be exported from and imported into other modules:

```
Some-module oid DEFINITIONS ::=
BEGIN
EXPORTS Some-type, some-value;
...
END
```

and in another module:

```
Another-module DEFINITIONS ::=
BEGIN
IMPORTS Some-type, some-value FROM Some-module oid;
...
END
```

If the EXPORTS statements is omitted, every type and value is exported. Any export can be prohibited by using an EXPORTS statements with an empty list. The *oid* is optional, but as those *object identifiers* are globally unambiguous, this ensures that the reference does not conflict with another module with the same name. Object identifiers are explained in Section 2.1.1.9.

This name scoping model is quite different from that of the C language. C does not have a concept of module names, all values and types live in the same namespace. In ASN.1 however, conflicting references may be resolved by prefixing the value or type reference with the module reference, e.g. Some-module.some-value vs. Another-module.some-value. In C++, the ASN.1 model can be emulated using nested classes or, using the probable new C++ standard as defined it the current draft [1], using a new scope, the namespace<sup>1</sup>. Snacc's C++ backend does use neither nested C++ classes nor C++ namespaces. The C++ classes get their name from the ASN.1 type name, with hyphens translated to underscores. Collisions among class names are resolved by tacking

---

<sup>1</sup>The scopes of current C++ are: local, function, file and class. [15, Section r.3.2]

numbers to the conflicting class names, e.g. the ASN.1 types `Some-Module.Some-Type` and `Another-Module.Some-Type` in C++ become `Some_Type` and `Some_Type1`.

ASN.1 module, type and value names are called references. The names of components of ASN.1 structured types, of numbers of INTEGER and ENUMERATED types and of bits of BIT STRING types are called identifiers. The names' syntax is similar to that of C/C++ identifiers, with some exceptions:

- all names may be composed of ASCII letters, digits and hyphen.
- type and module references must start with an upper-case letter,
- value references and identifiers must start with a lower-case letter.

Snacc converts hyphen into underscore and appends digits to identifiers that conflict with reserved words in the target language. More details are discussed in Section 3.3 on page 20.

## 2.1 ASN.1 Types

ASN.1 has simple and structured types. In addition, it defines a number of so-called “useful types” using ASN.1 itself.

ASN.1 has no pointer concept. In C++ pointers serve different purposes, among other things they are used to express optional data and for the implementation of recursive data types. In ASN.1 optional data can be expressed as detailed in Sections 2.1.1.1 and 2.1.2.1. Recursive data structures fall into two categories, trees and graphs. (Trees are graphs as well, but graphs may be circular and trees may not.) Trees may be expressed using optional data (the absent ASN.1 object is equivalent to C++'s terminating NULL pointer). Graphs in ASN.1 are not possible (i.e. you have got to emulate them, e.g. using integers to identify the nodes and lists of pairs of integers for the edges).

### 2.1.1 Simple Types

Simple types are the basis of all type definitions. Most of ASN.1's simple types are similar to those of C/C++, but the types NULL and OBJECT IDENTIFIER are quite unique.

#### 2.1.1.1 NULL

Let's begin with ASN.1's NULL type. This is a rather strange type: it defines only one value, NULL. In BER this value is represented using zero data bits (it occupies two octets, one for the type id and another to indicate the zero length data field).

Snacc maps the NULL type into a class with no data members.

ASN.1 has not got any pointers, but in cases where the pointer is used to express an object that may be present or absent, the NULL type can be used to get the same effect. The ASN.1 type definition

```
Optional-Integer ::= CHOICE { INTEGER, NULL }
```

defines a type that may contain an integer or that may be empty. In C++ this would be expressed by an `int *` that may be `NULL` to express the integer's absence. Snacc maps CHOICE types into unions containing pointers (see Section 2.1.2.3), and so C++ programmers get what they want: an `int *`. (Another method to get a C++ pointer is to define a SET or SEQUENCE component as OPTIONAL, see Section 2.1.2.1.)

### 2.1.1.2 BOOLEAN

ASN.1's BOOLEAN type has got two values, TRUE and FALSE.

The C language and the first two versions of the C++ language do not have a boolean type, logical values are expressed by integral values where 0 expresses false and every other value expresses true. The upcoming new C++ standard [1] has a `bool` built-in type, with `true` and `false` to denote its values. Snacc uses the new type where available (gcc 2.6 has it got already) and defines a look-alike otherwise.

### 2.1.1.3 INTEGER

The ASN.1 INTEGER type corresponds to the mathematical set  $\mathbf{Z}$  and poses no limits on the value range. Snacc, however, maps the ASN.1 type to a 32 bit integer type. For most applications this will be adequate, and for other uses Snacc has been extended using either some larger compiler supported integer type (e.g. `long long int`), or by using some arbitrary precision library.

Integer values may be given names as in

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

```
one-color RainbowColor ::= indigo
another-color RainbowColor ::= -30
```

but often this sort of type is restricted to the named values and in this case would be better expressed using an ENUMERATED type, see below.

In ASN.1 INTEGER types may be restricted to values or subranges or lists thereof, but when compiled with Snacc, this shows no effect in the resulting code. The 32 bit type used by Snacc for its INTEGER type mapping in ASN.1 could be expressed as:

```
Snacc-Integer ::= INTEGER (-2147483648 .. 2147483647)
```

### 2.1.1.4 ENUMERATED

ASN.1's ENUMERATED type looks very similar to the INTEGER type with numeric values given a name, but it differs from the INTEGER type in two ways:

- the values are restricted to those named.

- only the identifiers may be used for value denotation, their numbers are not allowed.

Sign ::= ENUMERATED { negative(-1), zero(0), positive(1) }

thumb-up Sign ::= positive

ASN.1's ENUMERATED type is very similar to the enum types of C and C++. Snacc maps every ENUMERATED type into a class containing an enumeration.

### 2.1.1.5 REAL

The REAL type comprises almost arbitrarily large floating point numbers (in BER up to  $\sim 2^{2^{2044}}$ ), plus the special values  $\pm\infty$ . Snacc maps it into a double which on most architectures is 64 bits large (1 bit sign, 11 bits exponent and 52 bits mantissa) and may express a range of approximately  $\pm 10^{\pm 308}$  with some 15 significant decimal digits.

Value denotation in ASN.1 is a little difficult to read, as the numbers are written as triples { Mantissa, Base, Exponent }:

e REAL ::= { 271828182845904523536, 10, -20 }

The Base may be 2 or 10. The resulting value is  $M \times B^E$ . The three values zero and  $\pm\infty$  have to be denoted as, 0, PLUS-INFINITY and MINUS-INFINITY, respectively.

Again, in ASN.1 subtyping may be used to restrict a type's value range, but shows no effect in Snacc's generated code.

### 2.1.1.6 BIT STRING

ASN.1's BIT STRING is a compact form for arrays of boolean values.

C and C++ have bit fields, but since it would be difficult to write BER encoding and decoding routines based on bit fields, Snacc's bit string types are based on a character array.

With the same notation that may be used to name INTEGER values, in BIT STRING types bit positions may be named. Numbering starts with position 0. Values may be denoted either binary ('00101'B), hexadecimal ('28'H) or as identifier list ({ flagA, flagC }) listing the "1" bits.

Example-Flags ::= ENUMERATED { flagA(2), flagB(3), flagC(4) }

Bit strings may be restricted in size, but again, the Snacc compiler ignores this.

### 2.1.1.7 OCTET STRING

The ASN.1 OCTET STRING represents arbitrarily long sequences of octets (8 bit bytes) and on almost any computer architecture maps easily into a char \* with accompanying length specification.

Values may be denoted in binary or hex; the following four values are all the same:

```

os0 OCTET STRING ::= '000010101110000111'B
os1 OCTET STRING ::= '000010101110000111000000'B
os2 OCTET STRING ::= '0AE1C'H
os3 OCTET STRING ::= '0AE1C0'H

```

OCTET STRINGs may be subtyped by restricting their length, but Snacc at present does not care.

### 2.1.1.8 Character String

Character string types can be used to represent textual information. ASN.1 defines eight character string types, two with alternative names:

|                                 |   |
|---------------------------------|---|
| NumericString                   | 0–9 <i>space</i>  |
| PrintableString                 | a–z A–Z 0–9 ' " ( ) + , - . / : = ? <i>space</i>                      |
| TeletexString<br>(T61String)    | character set as defined by CCITT Recommendation T.61                 |
| VideotexString                  | character sets as defined by CCITT Recommendations T.100, T.101       |
| VisibleString<br>(ISO646String) | visible characters of IA5 (international version of ASCII) plus space |
| IA5String                       | all characters of IA5   |
| GraphicString                   | all registered graphic characters plus space                          |
| GeneralString                   | all registered graphic and control characters plus space and delete   |

Character string values are denoted by their printed form, enclosed in double quotes (""). Double quotes inside the string are doubled, e.g. "double quote: >""<".

Character strings may be subtyped by restricting their length or alphabet, for example:

```

ISBN ::= PrintableString
      (SIZE (13))
      (FROM ("0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"|"|"-"|"X"))

```

Snacc disregards any subtyping and handles all characters string types like OCTET STRING. In particular, Snacc prints values in hexadecimal notation ('...H) instead of the correct string notation ("...").

### 2.1.1.9 OBJECT IDENTIFIER

Object identifiers are globally unique identifications for information objects like modules, abstract syntaxes like ASN.1, transfer syntaxes like BER, from small objects like content types to algorithms and full blown standards.

Object identifiers are build from a string of at least two non-negative integers.

An object identifier points to a node in the *object identifier tree* (OIT). This tree is an n-ary tree where every node is identified by a non-negative number. The object identifier is simply the sequence of numbers that leads from the root to the node, for example:

```
telephoneNumber OBJECT IDENTIFIER ::= { 2 5 4 20 }
```

To make those ids more readable, they can be given names. The names of the first few levels are well known (defined in ASN.1 itself) and the names of deeper levels can be introduced by following the name with its number in brackets:

```
telephoneNumber OBJECT IDENTIFIER ::= { joint-iso-ccitt ds(5) attributeTypes(4) 20 }
```

The first sequence element may be an object identifier itself, and the successive elements may be references to non-negative integers:

```
directory OBJECT IDENTIFIER ::= { joint-iso-ccitt ds(5) }
phoneAttr INTEGER ::= 20
telephoneNumber OBJECT IDENTIFIER ::= { directory attributeTypes(4) phoneAttr }
```

Object identifier values are given meanings by registration authorities. Each registration authority is responsible for all sequences beginning with a given sequence, in other words: for their subtree. A registration authority typically delegates responsibility for subsets of the sequences in its domain to other registration authorities.

## 2.1.2 Structured Types

Structured types are containers for other simple or structured types. They may be defined by listing their components or by referencing other structured types in different ways as shown below.

Components of structured types may be tagged, and in cases where the BER encoding would be ambiguous they must be tagged. Since there are many different cases where ambiguities may arise, it is good practice to tag all components of structured types, especially CHOICE types. In the below File example [0] is the name component's tag. A tag number may (but should not) be an arbitrarily large non-negative integer value (or a reference to such a value).

### 2.1.2.1 SET and SEQUENCE

A SET is an unordered, a SEQUENCE an ordered collection of components. Components may be defined as OPTIONAL, or they may have a DEFAULT value (in which case the component may be left unspecified in value notations, but it is nevertheless present).

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

Snacc maps SETs and SEQUENCEs into the same type, a C++ class. OPTIONAL components and components with a DEFAULT value are referenced through a pointer that may be NULL to denote the absent member.

**2.1.2.2 SET OF and SEQUENCE OF**

SET OF is an unordered collection of one single component type, and SEQUENCE OF is an ordered one. The number of components may be restricted as in

```
Pair ::= SET SIZE (2) OF Type
```

but Snacc makes no effort to enforce this.

Snacc implements both types as a double linked list.

**2.1.2.3 CHOICE**

CHOICES are what in C/C++ is known as union. A CHOICE takes exactly one of its subtypes.

Here is an example where tagging is necessary:

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

Snacc maps CHOICE types into C++ classes with an unnamed union of pointers for the possible components and an enum to memorize the chosen member.

**2.1.2.4 ANY and ANY DEFINED BY**

The ANY type can take values from any simple or structured type.

ASN.1 has a construct ANY DEFINED BY as in

```
AttrValue ::= SEQUENCE
{
  attrType INTEGER,
  attrValue ANY DEFINED BY attrType
}
```

where the attrType specifies which type of value attrValue has got to take. The specifier may be either an integer or an object identifier. The specifier to type mapping cannot be specified in ASN.1.

Snacc supports ANY DEFINED BY in two ways:

- by supplying the necessary hooks: two stubs, an enumeration type and a constructor, that have to be filled out by the programmer
- if the Snacc supported SNMP macro OBJECT-TYPE is used, Snacc fills the enumeration and constructor itself.

### 2.1.3 Useful Types

The ASN.1 specification defines some types using ASN.1 itself.

These are the types

**GeneralizedTime** and **UTCTime** to denote date-and-time specifications

**EXTERNAL** to embed arbitrary values from other abstract syntaxes

**ObjectDescriptor** to give human readable descriptions

Snacc defines these types in a file called `asn-useful.asn1` and has a special option `-u` after which this file may be specified. In this file, the character string types from Section 2.1.1.8 can be found as well.



## Chapter 3

# Snacc

This chapter is intended to provide a quick overview of Snacc, the ASN.1 compiler. An in-depth description of the compiler design and the supporting libraries can be found in Snacc's user documentation [13] which has grown to a document of some 200 pages.

Snacc compiles ASN.1 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 format. The name “snacc” is an acronym for “Sample Neufeld ASN.1 to C/C++ Compiler”. Snacc has been written by Michael Sample at the University of British Columbia (Canada). In 1993, Mike released two Snacc versions to the public, under conditions of the GNU General Public License.

Compiling ASN.1 into C is not a new idea but many other tools such as UBC's CASN1, ISODE's PEPY/POSY, and commercial tools either do not parse ASN.1 '90, produce slow encoders and decoders, generate C but no C++ code or are outrageously expensive. The aim of this tool is to provide an ASN.1 compiler that parses ASN.1 '90, produces efficient encoding and decoding routines and is freely available. Effort has been made to make the generated encoders and decoders relatively easy to fit into different software environments.

Some of Snacc's features include:

- parses CCITT ASN.1 '90 including subtype notation
- can compile and link inter-dependent ASN.1 modules (IMPORTS/EXPORTS)
- some X.400 and SNMP macros are parsed

**Note:** in this and all following chapters, an ellipsis (“...”) at the beginning of a file name path means “the top-level directory of the unpacked Snacc distribution”.

### 3.1 General Overview

Snacc consists of two main parts: the compiler and the libraries.

### 3.1.1 The Snacc Compiler

The Snacc compiler turns every ASN.1 module into a `.C` and a `.h` file. The compiler code can be further categorized into a core and a number of backends.

Both the compiler generated functions and the backend functions serve the following purposes:

- allocate and de-allocate memory for in-core data.
- convert the BER encoded data into their in-core representation and vice versa.
- print a human readable presentation of a value into a `FILE *` for C or into an ostream for C++.

First, the compiler's ASN.1 parser reads the input files and transforms them into an internal representation, the parse tree. The parser is built using the usual compiler building tools, `lex` and `yacc`, or, more likely, its GNU counterparts, `flex` and `bison`. Then, the compiler performs a dozen passes. The compiler core traverses the parse tree to resolve references across modules and to normalize types. Errors are detected in many passes. Finally, when the type information is finished, a backend is called that dumps the internal representation into a number of files in the target language, C or C++. The C backend writes structures and functions, the C++ backend writes classes and member functions. The backends generate code for ASN.1's structured types and for simple types with named values.

Alternatively, the compiler core can generate type tables that can then be processed by type table functions to be found in one of Snacc's C runtime libraries.

### 3.1.2 The Snacc Libraries

For every target language, there is at least one library that defines the types and functions for ASN.1's built-in simple types, for ANY (DEFINED BY) and for some buffer management routines.

There are three different C runtime libraries that differ in their underlying buffer management routines.

The functions that process the type tables generated by the compiler can be found in the fourth C runtime library.

For C++, one buffer type has been implemented only, and there are two C++ runtime libraries: one with the additional metacode and Tcl interface code and one without any extensions that is basically the same as the one and only C++ library of Snacc 1.1.

## 3.2 Evolution from Version 1.1 to 1.2rj

The original author, Mike Sample, released two Snacc versions, 1.0 and 1.1.

Despite of its strengths, Snacc 1.1 has some deficiencies:

- the same configuration options have to be edited in five config files and some 20 makefiles.

- every tiny source code change in Snacc's C runtime library leads to three full recompilations of all the library's source files.
- the C++ code lacks destructors.
- the C++ code leads to object files and executables that are inflated by large amounts of duplicate code.
- it has got three almost identical copies of the C runtime library.
- the compiler binary, the runtime libraries and their include files, the type table utilities and the manual pages must be installed manually.

Release 1.2rj<sup>1</sup> of Snacc addresses these problems and adds some new features.

- The configuration process has been simplified (at least for the installer of Snacc ;-)) by the use of GNU autoconf. With the exception of a small number of flags (currently three) in `.../policy.h`, no files have got to be edited to configure Snacc.

The Snacc compiler and library C code have been written to support K&R C as well as ANSI C. The configuration script tries to find out whether your C compiler understands ANSI C.

- 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! With the new set of makefiles, only those files that need to be re-made are. Following usual conventions, the phony targets `depend`, `check`, `install`, `clean` and `clobber` have been added.

See Appendix B for more explanations of many of the makefile tricks.

The configuration script generates the file `makehead` which gets included at the beginning of all `makefiles`. It contains a lot of definitions used for making.

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.

A third file that is included by almost every makefile is called `maketail`. As the name suggests, it gets included at the makefile's end. It holds the rules that are common to all makefiles where C/C++ code is compiled.

- Snacc release 1.1 produces huge virtual inline functions.
  - Due to their size, by many compilers, these inlines wouldn't get inlined anyway.
  - Virtual functions do not 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).

---

<sup>1</sup>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.

Instead, compilers generate static functions in every .C file were the .h file is included! This inflates the .o files and executables real quick (several MBytes per executable).

These functions have been turned into normal functions.

- const qualifiers have been added to a lot of C++ member functions: Clone(), comparison operators, Print() and type conversion operators. This allows the functions to be used on constants, such as the objects representing the values defined in ASN.1 modules.
- 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. This, of course, is the main topic of this diploma thesis and will be explained in the following chapters.
- Some C++ compilers (gcc 2.6 for example) already have this probable new C++ standard's `bool` type built in. The configuration script automatically detects this. The Snacc code has been modified to always provide a `bool` type for compilation with both old and new compilers.
- Snacc has successfully been ported to Linux and Alpha OSF/1, and should be both byte order and 64 bit clean.

### 3.3 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.



## Chapter 4

# The Metacode

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.
- Given the name of a type, the metacode can be asked for the type's details: general ASN.1 type, names of values (ENUMERATED), names of bits (BIT STRING), component names (structured types), etc...
- Given the name of a type, the metacode is able to return a newly allocated object instance of that type.
- Given the address of an object, the metacode can be asked for the object's type information and the object's current setting.
- Given the address of an object and a string that identifies one of the object's components, the component's address can be retrieved.

The metacode is an extension to the Snacc compiler's C++ backend and the C++ runtime library. Snacc's C++ backend has got the information that is needed for the metacode: it knows the structure of the C++ classes—it is the backend's task to generate them—and the names of all types and their components are present in string form—the backend needs them to generate the C++ source files. The runtime library's metacode does not need any structural information, but the backend generates recursive function calls for the components of structured types and the runtime library has to provide the endpoints for these recursions.

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.

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

To enable snacc's metacode generator, you have to give it an additional `-meta` option, followed by the list of PDU types. The metacode can be made to use either the names of modules, types, components, named values and bits as they were defined in the `.asn1` files, or it can use the names after they have been modified to fit into the C++ language. The optional `-mA` option makes the metacode use the ASN.1 names, and the `-mC` option switches to the backend's names. Section 3.3 on page 20 explains the name mapping.

## 4.1 Implementation

For both the metacode and the Tcl interface, a number of functions has been added to the C++ runtime library and a lot of code has been added to the backend so it can generate the required additional objects and functions.

The code changes concentrate in the following source files:

In the compiler core `.../compiler/core/snacc.c` has been extended for snacc to accept and handle the additional command line arguments. The files `.../compiler/core/meta.[hc]` have been added, they contain some utility code.

In the C++ backend, the file `.../compiler/back-ends/c++-gen/gen-code.c` contains the routines that write the C++ classes and their member functions.

The directory `.../c++-lib/` contains the C++ runtime library code. Every simple ASN.1 type has its own pair of `.h` and `.C` files that contain the ASN.1 type's C++ class counterpart and the set of functions to allocate the class instances, encode the data and so on. A number of files has been added in the backend: Most of the metacode is implemented in `.../c++-lib/{inc,src}/meta.[hc]`, as are the Tcl interface functions that directly call the metacode functions.

Now let us have a look at how the metacode works.

The metacode is implemented in both the data classes and in some global data structures.

The code that has been added to the data classes describes the data class they are contained in. A “data class” is the normal C++ class that is the result of Snacc's ASN.1 to C++ mapping. A class that implements the additional meta information will be referred to as “description class”. The description classes' names end in `Desc`.

The global data structures allow generic programs to find all modules and their types without having to know any of their names before hand.

Every `.asn1` file contains an ASN.1 module. Snacc compiles every module into a pair of C++ source code files, a `.h` and a `.C` file. In this pair of source files there are the normal data class definitions and their functions. The metacode is added at two places:

1. Every data class gets one or two static data members and one or two virtual functions.
2. In every C++ source file pair there is an array of pointers. Every array element points to the one additional static data member that has been added to every data class.

For the metacode snacc generates one additional C++ source file: `modules.C`. It does not correspond to an individual module. This file contains the array that provides the single well-known entry point for generic programs. The array contains one element for every source file pair, or: every ASN.1 module. The elements contain the module name and a pointer to the source file pair's pointer array described above.

Let us get back to the data classes and look at their additional metacode in more detail.

Every data class gets a little bit of additional code: one or two static data members and one or two virtual function members. The data members are static, and therefore get instantiated exactly once per executable, not once per object instance. The metacode put into every C++ class is very similar:

- a static `_desc` member is always present. It is the object that is pointed to by every module's pointer array. The `_desc` member describes its data class. There is one description class for every *general* ASN.1 type. “general” is meant as: one description class for all INTEGER data classes, one description class for all SET data classes, etc...
- simple types with names (ENUMERATED, INTEGER, BIT STRING) get an additional static `_mdescs[]` 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 `_getdesc()` function. Its only purpose is to return the address of `_desc`<sup>1</sup>.
- structured types with members (SET, SEQUENCE, CHOICE) get an additional virtual `_getref()` function. This function provides the member name to member address mapping.

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
}
```

---

<sup>1</sup>This sounds as if virtual data members were a nice idea, and in fact they are. The C++ standards committees are currently discussing their addition to the C++ standard.

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

get turned into these two C++ classes<sup>2</sup>:

```
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 generated .h file, the following code is taken from the accompanying .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()
```

---

<sup>2</sup>In reality, three classes are generated: The SET OF type in Directory gets turned into an additional C++ class DirectorySetOf. You can find its definition on page 36.

```

{
    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 (Istrcmp (membername, "name"))
        return &name;
    if (Istrcmp (membername, "files"))
        return &files;
    return NULL;
}

#endif // META

```

The two ASN.1 types get turned into two individual C++ classes, File and Directory, but their `_desc` members point to two different instances of the same type, `AsnSetTypeDesc`.

In the rest of this section, the code will contain the functions the Snacc Tcl interface uses to access the metacode functions. The Tcl interface will be described in Chapter 6.

There are two C preprocessor macros that are used for conditional compilation: `META` for the metacode and `TCL` for the metacode's Tcl interface.

The type of the `_desc` member differs depending on the general ASN.1 type it describes. 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,
}

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;

#ifdef 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 4.1.6.

The `name` is either the type's name as used by the backend code or the type's given name as defined in the `.asn1` file. 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 only used as an index into the `typenames` array—the virtual function call mechanism obviates 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 4.1.4.

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`.

This was the description base class. The following section will describe the the data classes base class and an example that is derived thereof.

Every ASN.1 type is represented by a C++ data 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` routine that generates a new instance (not a copy) of the object that it is invoked on.
6. it has an assignment operator.
7. it has a content encode and decode method, `BEncContent` and `BDecContent`, that only deal with the content of the type their object represents.
8. it has a PDU encode and decode method, `BEnc` and `BDec`.

9. it has a top level interface to the PDU encode and decode methods: BEncPdu and BDecPdu. They present the simplest interface; they return true if the operation succeeded and false if an error occurred.
10. it has a print method, Print, a virtual function that gets called from a global <<-operator. It 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 above routines are not affected by the metacode. If the metacode has been enabled, the following additional functions are present:

11. 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, the following additional functions are present:

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

The `AsnType` class is defined as follows:

```
class AsnType
{
public:
    virtual ~AsnType();
    virtual AsnType *Clone() const;

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

    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 *membenames);
    #endif // TCL
    #endif // META
};

```

As an example of a class derived directly from `AsnType` and described by `AsnTypeDesc`, here is the definition of `AsnBool` (for `BOOLEAN`):

```

class AsnBool: public AsnType
{
protected:
    bool          value;

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

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

```

The other four data class types that are described by the description base class are very similar: Instead of the `bool` value, `AsnReal` (for ASN.1's `REAL` type) contains a double value, and instead of the `AsnBoolTypeDesc _desc` it contains an `AsnRealTypeDesc _desc`. For `AsnOcts` (`OCTET STRING`) and `AsnOid` (`OBJECT IDENTIFIER`) the situation is similar. `AsnNull` (for ASN.1's `NULL` type) does not define any value data member, the type's single `NULL` value can be represented using zero bits.

Some data classes define some more constructors, comparison operators and/or access functions. As their description is not necessary to understand the metacode, they are only mentioned here.

### 4.1.1 Named Values

Some basic ASN.1 types allow values to be named, namely INTEGER, ENUMERATED and BIT STRING.

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 symbols will often be exactly the same as their ASN.1 counterparts. The base data classes for ENUMERATED, INTEGER and BIT STRING are AsnEnum, AsnInt and AsnBits, respectively.

Each of the three accompanying description types contains an array `AsnNameDesc_nmdescs[]` listing the names and the number or bit position values. The array's address is given to the type description's constructor as last argument. The virtual function `getnames()` returns this array.

```

struct AsnNameDesc
{
    const char          *const name;
    const AsnIntType    value;
};

struct AsnNamesTypeDesc: AsnTypeDesc
{
    const AsnNameDesc    *const names;

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

    const AsnNameDesc    *getnames() const;

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

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

#ifdef 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 or the value's name as given in the `.asn1` file. The generated source code contains the respective counterpart printed in a comment.

The three base classes, `AsnEnum` (for ENUMERATED), `AsnInt` (for INTEGER) and `AsnBits` (for BIT STRING), look similar to the five types (`AsnBool` etc...) already discussed. In particular, they contain no `_mdescs[]` data member. The base types define no named values and therefore no names description is necessary.

One type to note is `AsnEnum`. It is derived from `AsnInt`, not `AsnType`. `AsnEnum` defines no value data member, it uses `AsnInt`'s value data member. It has to define its own `_desc` and `_getdesc()` members though, as otherwise, the description of an ENUMERATED type would be that of an INTEGER type.

### 4.1.2 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 constructor initializes its `_desc` member so that it references its `_mdescs` array. 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();

#ifdef 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();

#ifdef 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    // _ == l/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 or the component's name as defined in the `.asn1` file. The generated source code contains the respective counterpart printed in a comment. In cases where the ASN.1 component was not given a name, the back-end's generated 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 (a more complete example, but without the Tcl functions, was already shown above on pages 26–27):

```
class FooSet: public AsnType
    AsnInt          bar; // an example data member
    ... // a lot 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 *membname);
#endif // TCL
#endif // META
};
```

`_getref()`'s bool parameter `create` determines whether a non-existing member should be returned as NULL pointer or whether it should instead be allocated and its address be returned. This parameter is used by the Tcl interface's 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 6.

All OPTIONAL components in a SET or SEQUENCE are referenced by pointer. The constructor will set OPTIONAL fields to NULL.

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.)

A CHOICE type's C++ class contains an anonymous union to hold the components of the CHOICE. Anonymous (unnamed) unions allow to reference the choice components with just the field name of the component; this makes referencing the contents of a CHOICE the same as referencing the contents of a SET or SEQUENCE.

### 4.1.3 SET OF and SEQUENCE OF

Every SET OF and SEQUENCE OF type is implemented using a double linked list. The classes that implement those lists provide a lot of access functions for the usual

operations: inserting, deleting, iterating across the elements.

The list description behaves like that of an ASN.1 simple type—the description type is derived directly from the 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 *);

#ifdef 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.

The introductory example on page 25 contains a SET OF type. Its implementation is as follows:

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

public:
#ifdef META
    static const AsnListTypeDesc _desc;
    const AsnTypeDesc          *_getdesc() const;
    AsnType                    *_getref (const char *index, bool create = false);
#endif
#ifdef TCL
    int                        TclGetDesc (Tcl_DString *) const;
    int                        TclGetVal (Tcl_Interp *) const;
    int                        TclSetVal (Tcl_Interp *, const char *valstr);
    int                        TclUnsetVal (Tcl_Interp *, const char *indexstr);
#endif
#ifdef META
    DirectorySetOf() { count = 0; first = curr = last = NULL; }
    // ... a lot of other functions omitted, destructor, Clone, list access and encode/decode functions...
};
```

#### 4.1.4 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 would not have any chance to preserve the two types' different names and thus, this construct cannot be used. A new C++ class has to be defined instead.

Example: the following two ASN.1 type definitions...

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

... get mapped into these 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` type 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 may have another pdu flag value, its type and names array values are those of its reference type. Therefore, `AsnAliasTypeDesc`'s first three of the five functions 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.

#### 4.1.5 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 metacode for 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 [7], ANY and ANY DEFINED BY can be found in “Annex I: Superseded features”, Section 3: “The any type”.

#### 4.1.6 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 preceding 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 in these modules.

## 4.2 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.

### 4.2.1 Normal Operation

The metacode does not significantly affect the normal mode of operation. The static data members `_desc`, `_nmdescs[]` and `_mdescs[]` 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 negligible. 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; when the code is compiled for other purposes, this task is left to the programmer who instantiates the object.

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.

### 4.2.2 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. 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%).

### 4.3 Metacode vs. Type Tables

Snacc's type tables and the metacode are two different approaches to the same problem. While both are intended for similar purposes, namely to be able to write generic programs that can handle data structures unknown before hand, their design, implementation and capabilities are quite different.

The metacode is an extension of the normal C++ code. When Snacc is used to generate type tables, no files that have to be compiled are generated at all. Snacc instead generates type descriptions in form of a BER encoded file (the ASN.1 description of which can be found in an appendix of the Snacc user documentation and online in the file `.../asn1specs/tbl.asn1`). This type description file can then be loaded from C programs using type table library functions. The type table gets loaded with `LoadTblFile()`, and the BER encoded data can be decoded using `TblDecode()`. Similar to the metacode, the ASN.1 types are accessed through character strings, and therefore it should be possible to build interpreted ASN.1 interfaces using Snacc's type tables.

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

- target 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, but no interface has been implemented that allows to access them by their names. This interface can easily be added, see Section 6.5 on page 64.
- named values:
  - The type tables lack the named values defined for ENUMERATED and INTEGER types and the named bits defined for BIT STRING types.
  - + The metacode interacts fine with these names and makes them accessible to interpreted interfaces.

- compatibility to normal snacc code:
  - The C structures defined by `mkchdr` (a type table tool that generates `.h` files from types tables) and used by the type table encoding and decoding routines on the one hand and the C structures defined by `snacc`'s C backend on the other hand are quite different. Where the backend's structures generated for `SEQUENCE` types contain mandatory members by value, the type tables' structures contain only pointer members.
  - + The C++ classes defined with the additional metacode have not got any additional non-static data members. The same application code can be compiled with and without the metacode (and the Tcl interface code).



## Chapter 5

# Tcl and Tk

Tcl is a simple scripting language which the author, John K. Ousterhout, describes in his book titled “Tcl and the Tk Toolkit” [10]. 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 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 numeric operation, converts the resulting integer 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.

### 5.1 Programming in Tcl: Problems and Pitfalls

The rest of this chapter is intended as grab-bag for Tcl and Bourne shell programmers. Some sections will be referenced from sections of Chapter 7 where the editor’s implementation details will be discussed.

#### 5.1.1 The Input Parser

Tcl’s parsing paradigm is almost identical to that of lisp: first, the structure is parsed, then the first token is elected to be the command that gets called with the tokens collected. In other languages, the commands are part of the syntax and the input parser behaves according to the command tokens it recognizes. In Tcl and lisp it is easy to introduce new commands, in C and perl it is impossible. The only difference between Tcl and lisp: in lisp, a procedure call is enclosed in parenthesis, in Tcl it is delimited by newline or semicolon. In most languages, white space is used only to separate input tokens but is otherwise insignificant. Tcl is an exception.

Example: in C you can write

```
if
(cond)
{
    do_something;
} if (cond) { do_this_as_well; }
```

and if you put a “\$” before the conditions, the example becomes valid perl<sup>1</sup> code.

In C and perl, braces group commands, in Tcl, braces are quotes. The rather odd looking Tcl code

```
if 1 "puts {hello}"
```

makes perfect sense and prints “hello”, as do the following modified versions:

```
if 1 {puts {hello}}
if 1 {puts hello}
if 1 "puts hello"
if 1 "puts \"hello\""
if 1 {puts "hello"}
```

The code

```
if 1 {puts yes} if 1 {puts no}
```

prints “yes” but no “no”. The first `if` is called with six arguments (argument zero is the “if” itself). It evaluates the first argument, finds that it constitutes a boolean true, executes its second argument and stops.

```
if 0 {puts no} if 1 {puts yes}
```

finally results in an error message as the `if` command finds a boolean false as argument one, skips argument two and continues parsing with argument three where it stumbles across the second “if” where it expects either “elseif” or “else”. In Tcl, commands are separated by newline or semicolon and thus a proper way to write the example is:

```
if 0 {puts no}; if 1 {puts yes}
```

This prints our “yes” again.

Newlines can be escaped and escaped newlines can be used to get aligned opening and closing braces:

```
if {cond} \
{
    do_something
}
```

---

<sup>1</sup>Perl is a very popular interpreted language. It is a freeware tool that lends itself for text filtering as well as system management tasks.

### 5.1.1.1 Quoting

While Tcl programming should be as simple as shell programming, Tcl's different quoting mechanism may drive experienced sh-programmers nuts.

This will be illustrated with some examples (\$ is the sh prompt, % is the tclsh prompt): whereas in /bin/sh<sup>2</sup>,

```
$ args="a b c"
$ cmd0 $args
$ cmd1 "$args"
```

cmd0 is called with three arguments, in Tcl,

```
% set args "a b c"
% cmd0 $args
% cmd1 "$args"
```

both commands are called with only one argument! To call cmd0 with three arguments, you have got to write

```
% eval cmd0 $args
```

But beware! If the command contains spaces, you must quote them, so the canonical syntax to write this example really is:

```
% eval [list cmd0] $args
```

Pretty simple, isn't it? :-) On the other hand, once understood, this quoting mechanism is quite useful as it avoids hacks like sh's "\$@":

```
% set files [glob *]
% eval exec ls -l $files
```

How can the same effect be reached in a sh-script? Let us assume that we have got two files in the current directory, named "a b" and "c".

```
$ files=*
```

does not glob, as can be verified:

```
$ echo "$files"
*
```

We can use a trick to get the file list into the files variable:

---

<sup>2</sup>This is where the Bourne shell is installed on almost every UNIX system. Under Linux, /bin/sh is a link to GNU's Bourne-Again Shell, bash, but as far as this chapter's examples are concerned, it behaves like the original.

```
$ files=`echo *`
$ # (please note: quote → '` ← backquote)
$ echo "$files"
a b c
```

Fine, we now have got the file list, it seems. But wrong, we cannot use it:

```
$ ls -l $files
a not found
b not found
-rw-rw-r- 1 rj  0 Jul 24 15:57 c
$ ls -l "$files"
a b c not found
```

The solution:

```
$ set - *
```

the shell has globbed:

```
$ echo "$*"
a b c
```

and the files are separate:

```
$ for f do echo "$f"; done
a b
c
```

Now we have to quote each individual argument:

```
$ ls -l "$@"
-rw-rw-r- 1 rj  0 Jul 24 15:57 a b
-rw-rw-r- 1 rj  0 Jul 24 15:57 c
```

Some older Bourne shell versions have a bug that makes them turn an empty argument list into *one empty argument* instead of the correct *no arguments*. Therefore, the canonical form for our example really is:

```
$ ls -l ${1+"$@"}
-rw-rw-r- 1 rj  0 Jul 24 15:57 a b
-rw-rw-r- 1 rj  0 Jul 24 15:57 c
```

Do you care for the perl equivalent? Anyway, here it is:

```
opendir (D, '.');
@files = grep (/^[^.]\/, readdir (D));
closedir (D);
system ('ls', '-l', @files);
```

### 5.1.2 Non-orthogonal commands

The simplicity of the input parser lets you easily predict what a Tcl command is called with as arguments. But some Tcl commands call the parser again to evaluate their arguments and some do not. In most cases they have to do so to execute conditional code, but at some places they are merely trying to offer you a service where it is not strictly necessary.

A possible trap: the first argument to Tcl's `if` command will be evaluated as an expression, the first argument to `switch` will not! While

```
if {[some_command ...] == {abc}} {...}
```

is a reasonable thing to do, the similar looking

```
switch {[some_command ...]} { abc {...} }
```

will not show the expected result. It must be written as

```
switch [some_command ...] { abc {...} }
```

On second sight the reason for `if`'s behaviour becomes clear: it is the same as the `while` command's. The `while` command must evaluate the condition itself in order to get the usual loop semantics where the entry condition has to be evaluated again and again—the input parser, when given an unquoted condition, would evaluate the condition only once!

```
set i 0
while [expr $i != 1] \
{
  set i 1
}
```

loops forever as the `while` command is called with “1” as its first argument. The brackets in `while`'s first argument must be quoted to defer their evaluation. For `if` commands the difference is small, as it does not constitute a loop.

```
if [command] ...
```

and

```
if {[command]} ...
```

will bring the same result most of the time, that is when `[command]` and `[expr [command]]` result in the same boolean value.

An attempt was made to write some clean code:

```
set quittext {...}
set savetext {...}
set canceltext {...}

set buttons [list some combination of the above texts]

switch [lindex $buttons [eval [list tk_dialog ...] $buttons]] \
{
  $savetext \
  {
    if {...} ...
  }
}
```

```

    }
    $canceltext \
    ...
}

```

This does not work because the `$savetext` and `$canceltext` in the `switch` command's second argument do not get evaluated. The statement was rewritten to read

```

switch [lindex $buttons [eval [list tk_dialog ...] $buttons]] \
"
    $savetext \
    {
        if {[...]} ...
    }
    $canceltext \
    ...
"

```

but after realizing that now every bracket had to be escaped to defer its evaluation

```

switch [lindex $buttons [eval [list tk_dialog ...] $buttons]] \
"
    $savetext \
    {
        if {\[...\]} ...
    }
    $canceltext \
    ...
"

```

This code does not look as clean as it was intended to be. It was rewritten to have a fixed list of fixed button texts:

```

switch [lindex {save discard cancel} [tk_dialog ... Yes No Cancel]] \
{
    save \
    {
        if {[...]} ...
    }
    cancel \
    ...
}

```

It is not too uncommon to enclose command text in quotes ("`...`") instead of braces (`{...}`): Tk callbacks are often specified this way.

```

button .b -command {some_proc $some_var}
button .b -command "some_proc $some_var"

```

In the first line, `some_proc` would be called with the literal `$some_var`. This variable may be a local variable of the callback installer, inaccessible to the callback procedures, and therefore the second line must be used. The second version invokes the

callback procedure with the contents of the variable. The callback will be called with the contents split into words. To guard against that, the argument has to be quoted. Quoting with quotes (`"..."`) or braces (`{...}`) may work most of the time, but it is a good idea to use the canonical way:

```
button .b -command "some_proc [list $some_var]"
```

### 5.1.3 The Trace Mechanism

In the ASN.1 editor, the ASN.1 data is kept in two data structures: the C++ object contains a binary representation and the graphical user interface requires a Tcl typical string representation. When the ASN.1 data that is displayed in a Tk widget is changed, the C++ data structure has to be updated. Tcl's trace mechanism is an ideal way to do this: most Tk widgets operate on global variables that can be traced. Whenever the Tk widget changes its variable, a Tcl procedure gets called.

At some point during the implementation the trace mechanism appeared not to work. The reason was that the procedure that is called for a variable trace does not get the variable's canonical name. If a procedure has defined an alias for the variable using the `upvar` command and installs a trace using the alias, the trace function will be called with the alias, not the variable's true name.

The problem is that an alias that refers to a global variable is not automatically global as well and the trace procedure may generate an error when it tries to access the variable. In other constellations the trace procedure will not get called at all.

A simplified example shall illustrate the different (missing) possibilities.

The following code contains some variable initializations, the trace procedure `pvar` and five different trials to use it. The `pvar` procedure is essentially the same as in the Tcl book [10, page 133]. The code that handles array variables has been removed (the example does not use any array variables) and `upvar` is used to reference the variable in the global scope instead of the calling procedure's scope (Tk callback's have to use the global scope, this is explained in the below name space section).

```
set x 0
set x0 0
set x1 0
set x2 0

proc pvar {varname element op} \
{
    upvar #0 $varname var
    puts "variable $varname set to $var"
}

trace variable x w pvar
set x 1
trace vdelete x w pvar
set x 2

upvar #0 x y
trace variable y w pvar
```

```

set y 3
trace vdelete y w pvar
set y 4

proc t0 {} \
{
    upvar #0 x0 z
    trace variable z w pvar
    set z 5
    trace vdelete z w pvar
    set z 6
}

proc t1 {} \
{
    upvar #0 x1 z
    trace variable x1 w pvar
    set z 7
    trace vdelete x1 w pvar
    set z 8
}

proc t2 {} \
{
    global x2
    upvar #0 x2 z
    trace variable x2 w pvar
    set x2 9
    set z 10
}

if {[catch t0 msg]} {puts "caught t0: $msg"}
if {[catch t1 msg]} {puts "caught t1: $msg"}
if {[catch t2 msg]} {puts "caught t2: $msg"}

```

When the code is run it prints:

```

variable x set to 1
variable y set to 3
caught t0: can't set "z": can't read "var": no such variable
variable x2 set to 9
caught t2: can't set "z": can't read "var": no such variable

```

The first line shows the desired effect.

The second line demonstrates that `pvar` gets called with the alias `y`, not the canonical name `x`. The example functions because the alias is introduced in the global scope where `pvar` expects it.

Line 3: `t0` cannot work as `x0`'s alias `z` is not globally known.

The line missing between lines 3 and 4: In `t1` no error occurs but the trace is not triggered.

Lines 4 and 5: `t2` demonstrates that the only way the `trace` command is to be used is with the variable's canonical name: `x2` can be set.

### 5.1.4 Name Space

Tcl lacks a file scope for procedures and especially variables.

In C you have got the file scope and functions can be hidden by making them static. C++ has the additional class scope and functions can be hidden therein. Variables can likewise be hidden, so that they are accessible from one module or from some set of class methods only. In Tcl there is nowhere to hide, your only escape is to give your procedures and variables unique names that will hopefully not collide with your own and with other programmer's procedure and variable names.

The usual strategy is to build procedure names with a module name as a prefix, for example `tk_dialog` is the name of a Tk utility, a procedure that implements a dialog box.

When programming X in C using Xt, the X Toolkit Intrinsic, you can give your callback functions the address of a hidden variable. Every callback procedure has an Xt-Pointer `client_data` parameter for this purpose. The variable may be dynamically allocated and may not even have any name at all.

It is impossible to get rid of global variables using Tk. Tcl has got local variables, and Tcl procedures can access the local variables of their calling procedures by means of the `upvar` command, but since Tk widget callback procedures get called from the Tk event scheduler, not from the procedure that installed the callback, access to the installer's local variables is lost. A workable solution to this problem: a mechanism that generates unique variable names. The callback procedures get this variable name as argument. Both the callback installer and the callback function declare the variable as global.

If more variables are needed, one can use more variable names or one can use one single variable as an associative array. Using more variable names, a lot of callback procedures would have to be called with a long argument list and every time another variable is introduced, the code would have to be adjusted at a large number of places. A good way to lose oversight. Associative arrays are far easier to maintain: only one variable stub, all procedures are called with a steady number of arguments, and one can deallocate all bundled variables at once. Associative arrays are implemented using hash tables and are as efficient as ordinary variables.

The recipe to get individual variables or widget names is simple: use a counter to append a number to a unique name stub and a small loop to test the variable's or widget's existence. This is an excerpt of the Snacc editor's Tcl script:

```
# the counter:
set #file 0

proc new_file {handle} \
{
    global #file

    # the test loop:
    while {[wininfo exists [set toplevel .[set fileref file${#file}]]} \
    {
        incr #file
    }
}
```

```
# declare the variable as global:
upvar #0 $fileref file

# use it as associative array:
set file(handle) $handle

...
```

## 5.2 Conclusion

As soon as the Tk module for Perl5 stabilizes, it should be considered to stop writing new tools in Tcl that are not guaranteed to stay small. Tcl is nice for simple tasks, but programs tend to grow and parts thereof get extracted for reuse. This is where Tcl shows its weaknesses.

Perl scripts get parsed once at startup into an internal form. If the compilation detects any errors, the script is not executed. In Tcl, many syntactic errors go unnoticed until the offending particular piece of code is first executed.

The perl compilation step is very fast, and the runtime execution of Perl scripts is many times faster than that of Tcl, speed differences of 10:1 are common.

Perl is able to handle binary data, Tcl stops at the first NUL byte.

## Chapter 6

# Tcl Interface

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

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 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, the “snacc subcommands” will be referred to 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
pdu->foo->bar = 42;
```

... and...

```
# this is Tcl code
snacc set {$pdu 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 4 is for.

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

## 6.1 The `snacc` Tcl command

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

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}`.

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.

The editor user should not be able to crash the program. The metacode and the Tcl interface code have been made quite 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.

Examples of how many of the commands can be used are given in Section 6.2 starting on page 59.

The syntax of the below command descriptions is that introduced by Tcl's author; it is used in the Tcl/Tk book [10], the online manual pages and in the documentation of countless Tcl extensions. In particular, optional command arguments are enclosed in question marks<sup>1</sup>. Literal strings are set in *upright typeface*, variable arguments are set in *oblique typeface*. Tcl's `set` command for example has the following synopsis:

```
set varname ?newvalue?
```

Valid uses therefore are:

```
set x 3
set x
```

### 6.1.1 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 ASN-Type

---

<sup>1</sup>Brackets have been used to denote optional arguments in UNIX documents for decades, but Tcl uses brackets for command substitution.

- may be associated with an external file in the file system

Three of the commands below allocate new files and must be given a content type specification. This *type* has to be denoted as one argument, a Tcl list with two elements, module and type. If the type is unknown, an error is returned and Tcl's `errorCode` variable is set to `SNACC ILLTYPE`.

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*. For *type* see the notes at this sections beginning. No external filename is associated with this file. The command returns a file handle to be used for other file commands and as first path component for commands that operate on the file's contents (described below).

**snacc open type filename ?flags? ?mode?** Open a file and read and decode its contents. For *type* see the notes at this sections beginning. 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. Upon success the command returns the empty string.

**snacc read file ?type filename?** without the *filename*, rereads the file from its old place; otherwise opens *filename*, reads its contents which are expected to be of the given *type* into *file* and closes it. For *type* see the notes at this sections beginning. 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 integer value returned by the Snacc routines' error handling code.

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 EOBUFF`.

**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 info 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 chosen 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.

## 6.1.2 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.)

### 6.1.3 Operations on Contents and Structure

Finally, the 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  $0 \dots n-1$  to address existing elements and it must be in the range  $0 \dots n$  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 86) `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. For the individual types the values returned are:

**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”s and “1”s, 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** The value is returned as a pair, the choice member chosen and its value.

**SET** and **SEQUENCE** The value is 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** The value is returned as a list of values.

**snacc set path value** sets the subtree identified by *path* to *value*. For the the different types, the values are expected in the following formats:

**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”s and “1”s 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 **SET 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`.

Tk's example where one has to set widget commands to `{ }` to delete them was not followed. 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!

## 6.2 An Example Session

The following example session shall illustrate the `snacc` commands usage. (This is an example session for editor programmers, editor users do not have to type any

of these commands.) It assumes that the editor example files `edex0.asn1` and `edex1.asn1` (see Appendix C on page 85) have been compiled into a binary that has been linked with the necessary libraries.

The notation used is as in the Tcl book [10], i.e. “ $\Rightarrow$ ” indicates a normal return value and “ $\emptyset$ ” 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 86). The “sub” tells us that the type has not been marked as a PDU. The SET has the components “coord” and “color”; the “valid” tells us that they both are present (they always are because 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 got 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 works only for write access, on read access this is not possible:

```
snacc get "$file color rainbow"
∅ snacc get: illegal component "rainbow" in path
snacc get "$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 "$file color rgb" {{green 0} {blue 0}}
∅ mandatory member "red" is missing in list
snacc set "$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
∅ snacc get: no file named "file0"
```

## 6.3 Implementation

The Tcl interface has got two layers. The lower layer connects to the metacode, the upper layer implements the `snacc` Tcl command.

### 6.3.1 Lower Layer

In every data class, both the runtime libraries' and the backend generated, there are one or two virtual metacode functions, `_getdesc()` and `_getref()`. The Tcl interface accesses these methods through up to four additional virtual functions in the same data class and up to two virtual functions in the description class. These functions already could be seen in Chapter 4, "The Metacode".

These functions allow access to the type descriptions and to the data of objects of these types.

### 6.3.1.1 Generic Information Retrieval

The description classes are defined in `.../c++-lib/inc/meta.h`, their member functions (both the metacode and the low level Tcl interface functions that access them) are defined in `.../c++-lib/src/meta.C`.

Both the data classes and the description classes have a virtual `TclGetDesc()` function, and both get a `Tcl_DString *` as argument, that is a reference to a dynamic Tcl string. In both class hierarchies the functions have different implementations at appropriate places in the classes' inheritance trees.

`TclGetDesc()` of the description classes is used to get the steady description of a type. The data classes' `TclGetDesc()` is used to get the dynamic description of an instance of the type. The two functions are used to implement `snacc type` and `snacc info` (described in Section 6.1.2 on page 56, "Generic Information Retrieval" and Section 6.1.3 on page 57, "Operations on Contents and Structure", respectively).

`snacc type` calls the description classes' `TclGetDesc()`. This virtual function calls its base classes function via scope resolution (`AsnTypeDesc::TclGetDesc()`) and adds the type specific part itself.

`AsnTypeDesc::TclGetDesc()` accesses four virtual metacode functions `getmodule()`, `getname()`, `ispdu()`, `gettype()` (see `AsnTypeDesc` on page 27). The `TclGetDesc()` function of the derived classes usually calls the base classes function explicitly (`AsnTypeDesc::TclGetDesc()`) and a function called `TclGetDesc2()` to append the type specific part. `AsnNamesTypeDesc::TclGetDesc2()` for example appends the Tcl list for the named values of an `INTEGER` type and `AsnChoiceTypeDesc::TclGetDesc2()` appends the Tcl list with the `CHOICE` type's possible component selections.

`snacc info` calls a combination of `AsnTypeDesc::TclGetDesc()` (again, that is an explicit call to the description base classes function) and `var->TclGetDesc()` where `var` is a data variable (an instance of a data class). The data classes' `TclGetDesc()` adds the information that is specific to its variable object. The data classes for the ASN.1 simple types do not have their own `TclGetDesc()` function: the virtual function call mechanism calls `AsnType::TclGetDesc()` that returns `TCL_OK` without adding anything to the return string value. The data classes for the structured ASN.1 types define their own `TclGetDesc()` functions: `CHOICE` types append their current component's name to the return string value, `SET` and `SEQUENCE` types inform us about the presence or absence of their `OPTIONAL` members and `SET OF` and `SEQUENCE OF` types put the length of their list into the return string.

### 6.3.1.2 Operations on Contents and Structure

Data classes may have up to three functions that allow data retrieval and manipulation. These functions, `TclGetVal()`, `TclSetVal()` and `TclUnsetVal()`, are defined where the other member functions of their data class are implemented: `.../c++-lib/src/asn-*.C` for the runtime library and in `.../compiler/back-ends/c++-gen/gen-code.c` for the backend generated data classes.

They are typical Tcl functions:

- They get a `Tcl_Interp *` as first argument. This interpreter reference is used to put the result string in, and to set the interpreter's `errorCode` variable.

- The functions return an integer argument, either `TCL_OK` or `TCL_ERROR` to indicate success or failure, respectively.

For the ASN.1 types, the functions' implementation is very straight forward. For example, `AsnInt::TclGetVal()` uses `sprintf()` to turn the numeric value into a string value. To do the reverse, `AsnInt::TclSetVal()` calls the metacode function `getnames()` (see “Types with Names”, Section 4.1.1 on page 32) and compares the value string with the name strings. If it finds a match, it takes the name's value, otherwise it calls `Tcl_GetInt()` to convert the integer string to the integer value. The functions for the other simple types are similar. The ASN.1 simple types define no `TclUnsetVal()` function: the base classes function will be called which will return an error.

For the structured types, the three functions look a little different. `TclSetVal()`'s argument usually is a Tcl list. The functions check this list, for `SET` and `SEQUENCE` this list has to contain all non-`OPTIONAL` components, and recursively call to their components' functions. The metacode's `_getref()` function is used here (see “Types with Members”, Section 4.1.2 on page 33).

### 6.3.2 Upper Layer

The upper layer of the Tcl interface comprises two C++ classes, `SnaccTcl` and `ASN1File`, and some glue code. 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 `ASN1File` class is a container for a pointer to an ASN.1 type's C++ object and for an optional external file name and its file descriptor. `ASN1File` objects correspond to the handle returned by `snacc create` and `snacc open` that were described in “File Commands”, Section 6.1.1 on page 54.

The `snacc Tcl` command is implemented as a C++ class, `SnaccTcl`. The `snacc` sub-commands are implemented as member functions. Some of the functions call the lower layer Tcl interface functions, some access the metacode's information directly. Other functions invoke systems calls and interfere with the `ASN1File` class member functions.

The glue code initializes the Tcl interface. Upon startup, the `SnaccInit()` function is called from Tk's `main()` as explained below. `SnaccInit()`'s first task is to allocate a `SnaccTcl` object. The constructor traverses the metacodes module and type descriptions arrays and builds Tcl hash tables for easier access. The constructor allocates another Tcl hash table for the `ASN1File` objects.

The upper layer of 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 upper layer's second task is to introduce the `snacc Tcl` command to the Tcl interpreter. This step is simple when it is executed, but the implementation that forces the right functions to be called is a little confusing: 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.

This main() calls Tcl\_AppInit() and Tcl\_AppInit() calls Snacc\_Init() that is defined in `.../c++-lib/src/tcl-if.C`. Snacc\_Init() installs the `snacc` command by calling Tcl\_CreateCommand().

## 6.4 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 and tree widget libraries.
- The preprocessor switches `NO_META` and `NO_TCL` in `.../policy.h` must not be set to non-zero.

## 6.5 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 85.)

- The Tcl interface does not provide symbolic object identifiers. Mapping numeric to symbolic OBJECT IDENTIFIERS 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.

The leading part of an OBJECT IDENTIFIER arc could be replaced (but is not).

## Chapter 7

# 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.

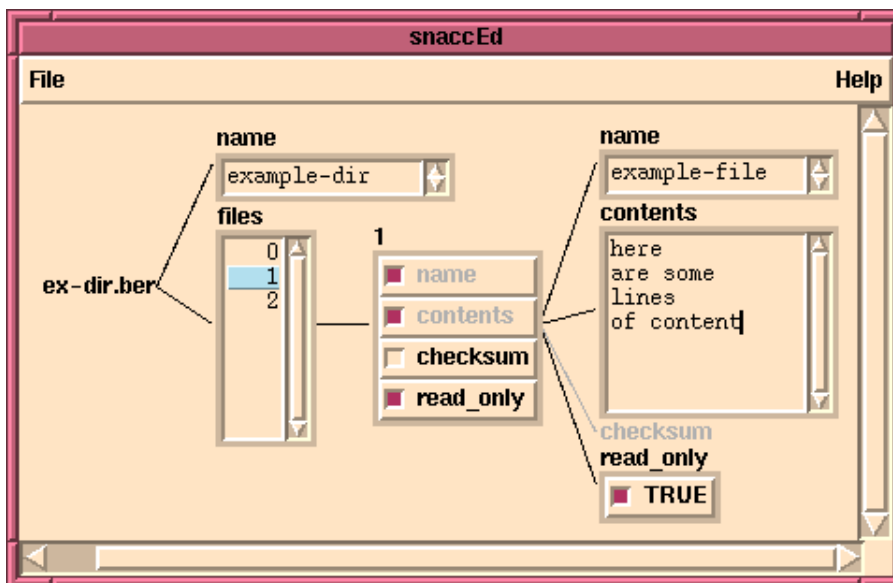


Figure 7.1: An Example Screen Shot

SnaccEd is combined from

- the usual Snacc BER encode and decode functions
- the metacode (described in Chapter 4 starting on page 23)
- the Tcl library
- the Snacc Tcl interface (described in Chapter 6 starting on page 53)

- 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`. This program is a Tcl interpreter that has the additional commands of the Tk widget set, of the tree widget and of the Snacc interface built in. 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. The Tcl/Tk interpreter has the name `wish`, for “windowing shell”, and consequently, the program that results from linking Snacc with Tcl/Tk is referred to as `snaccwish`.

This interpreter reads the Tcl scripts that implement the graphical Snacc editor. The Tcl scripts get loaded as necessary using Tcl's autoloading mechanism. A tiny Tcl script is used to start the interpreter and to load an initial editor script. This startup script will henceforth be referred to as `snacced`.

Both the script and the interpreter may be given other names, for example a `mheged` and `mhegwish` would be more appropriate names for an editor for MHEG data.

The combination of the interpreter and all the editor's Tcl scripts will be referred to as “SnaccEd”.

`snacced` can be called with a varying number of arguments:

- Three arguments are taken to be a module name, a type name and a file name. The according file is opened. Its contents, expected to be of the given type of the given module, are BER decoded.
- Two arguments are taken to be a module name and a type name. A file with that content type is created. It is not associated with an external file name.
- If `snacced` is called without command line arguments, it pops up its file and content type selection box (see Figure 7.2 on page 67) where the above to actions can be performed as well.

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

Except for `snacced`, the Tcl scripts are (or can be) always the same. They use 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 7.2 on page 67 for an example).

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

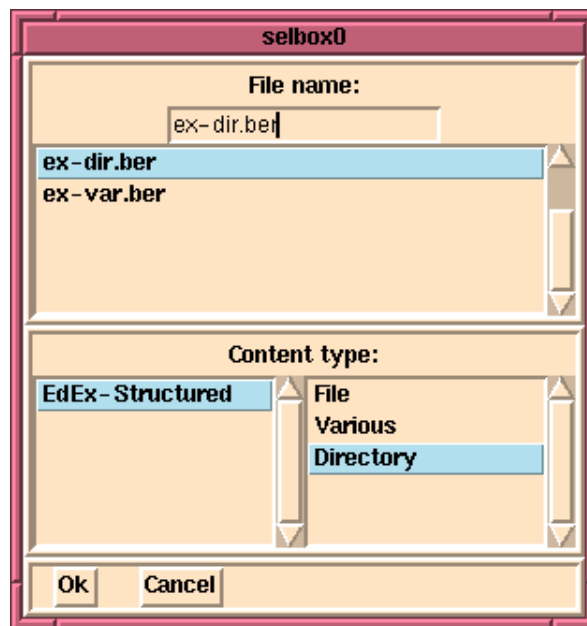


Figure 7.2: The File And Content Type Selection Box

## 7.1 Manipulating the Display

This section describes the pointer<sup>1</sup> operations that change the amount of information to be shown. (To change the file's contents, the node's content window has got to 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<sup>2</sup> 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 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 7.2 on page 71.)

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

**button 2** opens or closes the node, where “closed” means that only the node's 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.

<sup>1</sup>My pointer device is a mouse, but yours may be a trackball, a tablet, a joystick or something else.

<sup>2</sup>I will refer to the buttons by 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.

**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.

## 7.2 The Content Window

The content window, it may be opened beneath a node's name, looks and behaves differently for every content type. An example for every ASN.1 simple type is shown in Figure 7.3 on page 69. The ASN.1 input for the example can be found in Appendix C on page 85.

- 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 bindings have been changed to allow the input of minus sign and decimal digits only in addition to the usual control functions. Values can be given a name (this is similar to the ENUMERATED type), the list is displayed as above.
- Individual bits in a BIT STRING may be named. SnaccEd displays a list of buttons identifying those bits by 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 bindings have been changed to allow the input of "0" and "1" only in addition to the usual control functions.
- 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 (see Figure 7.5 on page 70). The X text selection to copy text between the text widget and e.g. an xterm can be used as well. The text can be dragged by pressing and holding button 2 on the text or by using the scrollbar.
- 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 7.4 on page 70.

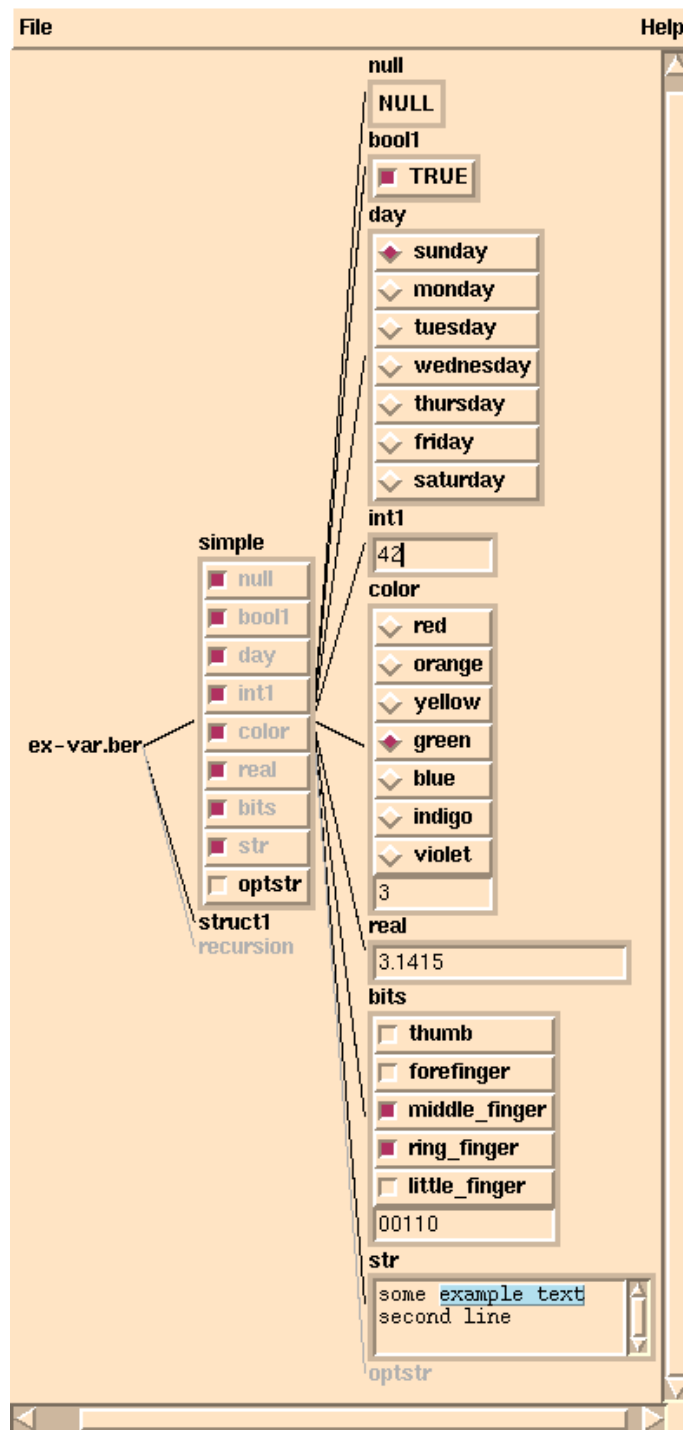


Figure 7.3: Content editors for ASN.1 Simple Types

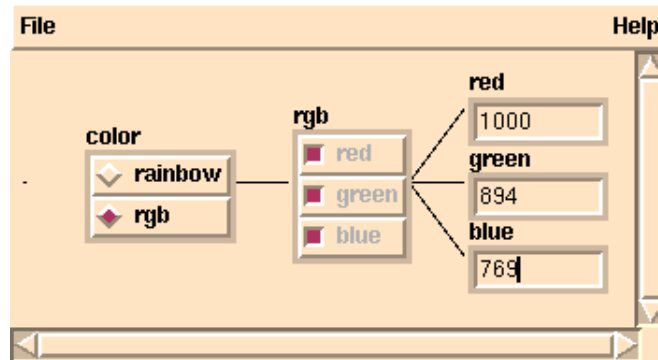


Figure 7.4: Content Editors for ASN.1 Structured Types

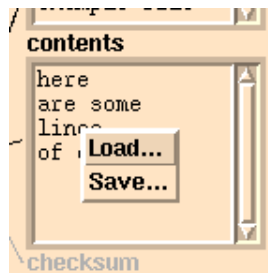


Figure 7.5: Popup for Import/Export of OCTET STRING Contents (based on the example displayed as Figure 7.1 on page 65)

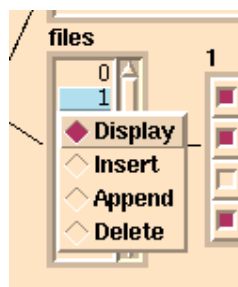


Figure 7.6: Popup for Action Selection for SET OF and SEQUENCE OF Types (based on the example displayed as Figure 7.1 on page 65)

- 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 7.1 on page 65 and the “rgb” in Figure 7.4 on page 70.
- 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 pops up a small menu where you can choose the action to perform when (with button 1) you click on a list element (see Figure 7.6 on page 70):
  - 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” shown in figure 7.1 on page 65 and in Figure 7.6 on page 70. The list can be dragged by pressing and holding button 2 on the list widget or by using the scrollbar.

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. The tree layout will adjust after releasing the button.

## 7.3 Building Your Own Editor

The Snacc user documentation contains a lengthy description on how to build an editor. This description is fairly long as it tries to catch all eventualities.

In short it boils down to this:

- Install Tcl/Tk, the tree widget and Snacc.
- Use snacc to compile the .asn1 files. The `-tcl` option enables the metacode and the Tcl interface code.
- Compile the Snacc generated source files.
- Link the object code with the appropriate libraries into the snaccwish.
- Generate the helper script that loads the rest of the editor scripts.

As a complete example, here is how to build this chapter's editor:

```
snacc -tcl EdEx-Structured.Directory -u asn-useful.asn1\
  edex0.asn1 edex1.asn1
c++ -DTCL -c edex0.C edex1.C modules.C
c++ -o edexwish edex0.o edex1.o modules.o -lasn1tcl\
  -ltktree -lOS -ltk -ltcl -lX11 -lm
make-snacc edex ./edexwish ../tcl-lib
```

After the editor has been started by executing

```
./edex EdEx-Structured Directory ex-dir.ber
```

it can be made to look like in Figure 7.1 on page 65.

## 7.4 Implementation

The Tcl scripts that implement the editor can be found in `.../tcl-lib/` and, after installation, in `${prefix}/lib/snacc/tcl/`. The interpreter loads them automatically when it is asked to run an unknown command.

Currently the editor consists of the following script files:

`bindings.tcl` contains the procedures that install the key bindings for the content editor widgets.

`help.tcl` contains the procedure that pops up a window to display a help text.

`selbox.tcl` contains the implementation of the file and content type selection box.

`uti.tcl` for general utility procedures.

`tkuti.tcl` for utility procedures in use with Tk.

`snacc.tcl` contains all other procedures that make up the editor.

The script that is generated by `make-snacc` and that is called to start the editor contains only three lines:

1. The interpreter name that fits to the ASN.1 descriptions is called to read this script.
2. The autoloading path is extended so that the interpreter finds the editor's generic Tcl procedures.
3. The editor's entry procedure, `snacc`, is called.

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

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

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 variable and toplevel widget names are generated in the code example on page 51: `new_file` is called with the file handle returned by `snacc create` or `snacc open`. `$fileref` contains the name or the array variable and this name is given to many procedures in their `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.

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 content type selection box, the help window 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.

### 7.4.1 Procedures

Changes to the editor's Tcl scripts will probably made more by extending the procedures, and far less by modifying the data structures. The data structures that are used throughout the editor scripts are described above, but the procedures are better described in the code itself. The code is quite easy to read and translating Tcl into English is probably less of a help than putting comments into the code. When the routines get changed, the documentation that was put here would no longer match.

## 7.4.2 Data Structures

The contents of an ASN.1 file can be seen as a tree—the data structure may be 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 7.7 on page 76 for an example), but as ASN.1 has not got any 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 file's concrete structure. This is what in Chapter 6 is always referred to as “*path*” argument to most *snacc* sub-commands. The *snaccpath* is a proper Tcl list.
- The *treepath* is very similar to the *snaccpath*. The *treepath* is used at a lot of places, for widget and variable names and for canvas item tags, all detailed in the below bulleted items. The structure of the *treepath* 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*.

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, the indices need to be incremented. The *treepath* is used for widget and variable names and for canvas item tags. 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, 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 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 and to retrieve the tag with a particular index from this 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”. Active node labels get the tag “valid-label”, the node labels of absent OPTIONAL components have the tag “void-label” and is inactive. Using this tag, for active nodes, the procedure `new_file` binds the three pointer button events to the call-back 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 editor's existence. No two items have the same value of this tag.
- Content editors are not simple canvas items. They are built 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`.

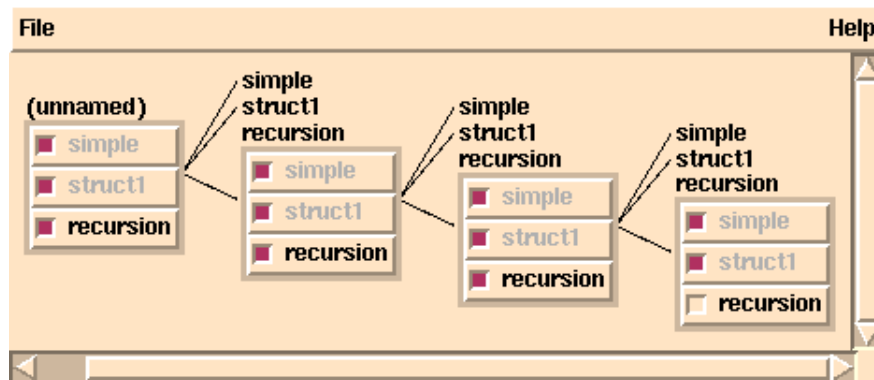


Figure 7.7: Data in a Recursive ASN.1 Type

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.

## 7.5 Future Work

The editor can be improved in a number of ways:

- It could be specialized to display certain content types in a more appropriate manner. For example, a text widget sometimes will not be the best way to display an OCTET STRING, pictures and audio data come to mind. The editor scripts could be extended
  - with an array variable that maps content types to procedure names.
  - so that `toggle_editor` checks the array variable, and calls the procedure if the array contains an entry for the current content type.

The additional procedures would have to adhere to some conventions:

- They would have to build their widget tree into a given frame widget.
- They have to write their widget's contents back to the binary C++ representation.
- If the file's contents is changed, the file's `modified` flag has to be set.

For example, an OCTET STRING that contains audio data would better be displayed in a button arrangement that resembles an audio recorder (“play”, “pause”, “stop”, “record”, etc...). Pressing the “play” button should send the data to an audio device.

- Upon startup, the generic editor displays the file's root node only. Key or button bindings could be added that “explode” the display, i.e. so that the full tree with all nodes gets displayed. This function should only be added for small trees or for a certain part of larger ones!

- The editor could be made to detect the file's content type automatically. In general this is not possible, but in cases where every content type starts with an OBJECT IDENTIFIER, this functionality can be added.
- The editor lacks an undo function. It can be implemented using a stack (implemented as a Tcl list) of callback commands. Every function that changes the file's contents would have to register a reverse operation. The undo function would simply evaluate successive list elements to implement an unlimited undo history. When the registering function is called from an undo callback, the history traversal continues, otherwise it is reset to the stack top.
- The editor does not keep a backup copy of the edited files.



## Appendix A

# Coding Tricks For Readability

One of our GLASS 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 more additional compilation conditions. Since the Tcl interface is only useful on top of the metacode, there are six ( $2 \times 3$ ) different combinations instead of eight ( $2^3$ ). Even these six combinations would have made the code very ugly. Please decide for yourself, here is an example with a function with only up to four arguments, the C++ backend's `PrintCxxCode()` has up to 17 arguments:

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

```

#else
FunctionName PARAMS ((b, c),      // (option B)
    TypeB b _AND_
    TypeC c)
#endif
#else
FunctionName PARAMS ((b),        // (no option)
    TypeB b)
#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, two macros would be necessary, one that put a comma before the argument, and one that puts it thereafter. 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 safely be used. It expands into code with brackets around it: `PROTO ((int a, char *b))` becomes `(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.

The `if_...` macros have to expand into code without brackets, for example `if_A (a COMMA b)` expands into `a, b`.

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 B

## Makefiles

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

### B.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<sup>1</sup> `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 `cv`s 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.

### B.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

---

<sup>1</sup>yes, System III, not System V R3

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;\
```

### B.3. COMPILING DIFFERENT LIBRARIES FROM ONE SET OF SOURCE FILES83

```
        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.

## B.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.

## B.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.



## Appendix C

# ASN.1 Files for the Editor Example

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

The first file contains some simple ASN.1 types:

```
-- 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
```

The second file contains some structured ASN.1 types. The example has been put into two files to demonstrate how one EXPORTS and IMPORTS types. The PrintableString in some types below is one of the so-called “useful types”; they do not have to be imported.

```
-- 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
```



# Bibliography

- [1] ANSI X3, Information Processing Systems. *Working Paper for Draft Proposed International Standard for Information Systems—Programming Language C++*, 28 April 1995.
- [2] J. Case, M. Fedor, M. Schoffstall, and J. Davin. *RFC 1157: A Simple Network Management Protocol (SNMP)*. University of Tennessee at Knoxville, Performance Systems International, Performance Systems International, and the MIT Laboratory for Computer Science, May 1990.
- [3] Comité Consultatif Internationale Télégraphique et Téléphonique. *Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1)*, 1988.
- [4] Comité Consultatif Internationale Télégraphique et Téléphonique. *Recommendation X.209: Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, 1988.
- [5] International Organization for Standardization. *Information processing systems—Open Systems Interconnection—Specification of Abstract Syntax Notation One (ASN.1)*.
- [6] International Organization for Standardization. *Information processing systems—Open Systems Interconnection—Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*.
- [7] International Organization for Standardization; International Electrotechnical Commission. *Information technology—Open Systems Interconnection—Abstract Syntax Notation One (ASN.1)*.
- [8] Burton S. Kaliski Jr. *A Layman's Guide to a Subset of ASN.1, BER, and DER*. RSA Data Security, Inc., Redwood City, CA, June 3, 1991.
- [9] Brian W. Kernighan and Dennis M Ritchie. *The C Programming Language*. Prentice-Hall, Inc., 1978.
- [10] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, 1994. ISBN 0-201-63337-X.
- [11] Eric Raymond, editor. *The New Hacker's Dictionary*. The MIT Press, Cambridge, Mass, London, England, 1991. ISBN 0-262-68069-6.
- [12] Jonathan Rees and William Clinger (Editors). *Revised<sup>3.95</sup> Report on the Algorithmic Language Scheme (Draft)*, March 1989.

- [13] Michael Sample and Robert Joop. *Snacc 1.2rj: A High Performance ASN.1 to C/C++ Compiler*, August 1995.
- [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 0-201-53992-6.

# Index

- .../, 17
- <<-operator, 30
- \_desc, 25, 38
- \_getdesc(), 25, 61
- \_getref(), 25, 34–36, 61
- \_mdescs[], 25, 33
- \_nmdescs[], 25, 32
- Abstract Syntax Notation One, *see*
  - ASN.1
- ANSI C, 19, 79
- ANY (DEFINED BY), 14, 38
- ASN.1, 7–15, 85
  - compiler, 17
  - Types, 9
    - ANY (DEFINED BY), 14
    - BIT STRING, 11
    - BOOLEAN, 10
    - Character String, 12, 15
    - CHOICE, 14
    - ENUMERATED, 10
    - EXTERNAL, 15
    - GeneralizedTime, 15
    - GeneralString, 12
    - GraphicString, 12
    - IA5String, 12
    - INTEGER, 10
    - ISO646String, 12
    - NULL, 9
    - NumericString, 12
    - OBJECT IDENTIFIER, 12
    - ObjectDescriptor, 15
    - OCTET STRING, 11
    - PrintableString, 12
    - REAL, 11
    - SEQUENCE, 13
    - SEQUENCE OF, 14
    - SET, 13
    - SET OF, 14
    - T61String, 12
    - TeletexString, 12
    - UTCTime, 15
    - VideotexString, 12
    - VisibleString, 12
    - useful types, 15
- ASN.1 module, 24, 85
- ASN1File, 63
- AsnAliasTypeDesc, 37
- AsnBits, 33
- AsnBitsTypeDesc, 32
- AsnBool, 31
- AsnBoolTypeDesc, 28
- AsnChoiceMemberDesc, 34
- AsnChoiceTypeDesc, 34
- AsnEnum, 33
- AsnEnumTypeDesc, 32
- AsnInt, 33
- AsnIntTypeDesc, 32
- AsnListTypeDesc, 36
- AsnMemberDesc, 33
- AsnMembersTypeDesc, 34
- AsnModuleDesc, 28, 38
- AsnNameDesc, 28, 32
- AsnNamesTypeDesc, 32
- AsnNullTypeDesc, 28
- AsnOctsTypeDesc, 28
- AsnOidTypeDesc, 28
- AsnRealTypeDesc, 28
- AsnSequenceMemberDesc, 34
- AsnSequenceTypeDesc, 34
- AsnSetMemberDesc, 34
- AsnSetTypeDesc, 34
- AsnType, 30
- AsnTypeDesc, 27
- Basic Encoding Rules, *see* BER
- BDec(), 29
- BDecContent(), 29
- BDecPdu(), 30
- BEnc(), 29
- BEncContent(), 29
- BEncPdu(), 30

- BER, 7, 65
- BIT STRING, 11, 32, 68, 85
- bool, 20
- BOOLEAN, 10, 31, 68, 86
- Bourne shell
  - quoting, 45
- CCITT X.208, *see* X.208
- CCITT X.209, *see* X.209
- CCITT X.400, *see* X.400
- Character String, 12, 15
- CHOICE, 14, 33, 68, 86
- Clone(), 29
- COMMA, 79
- data class, 23
- DEFAULT, 13, 86
- dependencies, 19, 81
- dependencies (circular), 81
- description class, 23
- ENUMERATED, 10, 32, 68, 85
- EXPORTS, 85
- EXTERNAL, 15
- ftp, 2
- GeneralizedTime, 15
- GeneralString, 12
- getmodule(), 38
- getnames(), 32, 38
- GLASS, 1
- globbing (Tcl and Bourne shell), 45
- GraphicString, 12
- IA5String, 12
- implementation
  - metacode, 24
  - SnaccEd, 72
  - Tcl interface, 61
- IMPORTS, 8, 85
- INTEGER, 10, 32, 68, 85
- ISO646String, 12
- ISO 8824, 7
- ISO 8825, 7
- K&R C, 19, 79
- make, 81
- make-snaccEd, 72
- makefile, 19, 81
- makehead, 19, 81
- maketail, 19, 81
- metacode, **23–41**
  - aliases, 37
  - ANY (DEFINED BY), 38
  - data class, 23
  - description class, 23
  - implementation, 24
  - named bits, 32
  - named values, 32
  - types with members, 33
- MHEG, 8
- MHS, *see* X.400
- modules.C, 25, 39
- name mapping, 20
- name space (in Tcl), 51
- named bits, 32
- named values, 32
- NUL, 68
- NULL, 9, 31, 68, 86
- NumericString, 12
- OBJECT IDENTIFIER, 12, 31
- ObjectDescriptor, 15
- OCTET STRING, 11, 31, 68, 86
- OPTIONAL, 13, 35, 86
- PARAM, 79
- Perl, 52
- PrintableString, 12
- PROTO, 79
- quoting
  - Tcl vs. /bin/sh, 45
- REAL, 11, 31, 86
- SEQUENCE, 13, 33, 71, 86
- SEQUENCE OF, 14, 35, 71, 74
- SET, 13, 33, 71, 86
- SET OF, 14, 35, 71, 74
- Snacc, **17–21**
  - compiler backend, 18, 24
  - compiler core, 18, 24
  - name mapping, 20
  - output file name, 20
  - runtime libraries, 18, 24
- snacc
  - close, 55, 61
  - create, 55, 60

- export, 56
- finfo, 56
- get, 58, 61
- import, 56
- info, 57, 60, 62
- modules, 56
- open, 55
- read, 55
- set, 58, 61
- type, 57, 60, 62
- types, 56, 60
- unset, 59
- write, 56
- SnaccEd, 65–77**
  - idlist, 74
  - implementation, 72
  - snaccpath, 74
  - tags (canvas items), 75
  - treepath, 74
- Snacclnit(), 63
- snaccpath, 57, 74
- SnaccTcl, 63
- snaccwish, 66
- SNMP, 8, 17
  
- T61String, 12
- Tcl, 43–52**
  - name space, 51
  - non-orthogonal commands, 47
  - parser, 43
  - quoting, 45, 48
  - variable tracing, 49
- Tcl interface, **53–64**
  - implementation, 61
- TclGetDesc(), 62
- TclGetDesc2(), 62
- TclGetVal(), 62
- TclSetVal(), 35, 62
- TclUnsetVal(), 62
- TeletexString, 12
- treepath, 74
- types with members, 33
  
- useful types, 15, 85
- UTCTime, 15
  
- VideotexString, 12
- VisibleString, 12
  
- X.208, 7
- X.209, 7
- X.400, 8, 17