ASN1C

ASN.1 Compiler User's Guide for Python

Version 7.4

Objective Systems, Inc.

January 2020

ASN1C: ASN.1 Compiler User's Guide for Python

Author's Contact Information. Comments, suggestions, and inquiries regarding ASN1C or this document may be sent by electronic mail to <info@obj-sys.com>.

Table of Contents

Chapter 1. Overview of ASN1C for Python

The ASN1C code generation tool translates an Abstract Syntax Notation 1 (ASN.1) or XML Schema Definitions (XSD) source file into computer language source files that allow typed data to be encoded/decoded. This release of ASN1C includes options to generate code in the following languages: C, C++, C#, Java, or Python. This manual discusses the Python code generation capabilities. The following manuals discuss the other language code generation capabilities:

- *ASN1C C/C++ Compiler User's Manual* : C/C++ code generation
- *ASN1C C# Compiler User's Manual* : C# code generation
- *ASN1C Java Compiler User's Manual* : Java code generation

Each ASN.1 module that is encountered in an ASN.1 schema source file results in the generation of an equivalent Python source file with the same name as the module with hyphens replaced with underscores and with extension '.py'.

There is also a set of classes that form the run-time component of the Python package. These classes provide the primitive component building blocks that are assembled by the compiler to encode/decode complex structures. They also provide support for managing message buffers that hold the encoded message components.

This release of the ASN1C Compiler for Python works with the version of ASN.1 specified in ITU-T international standards X.680 through X.683. It generates code for encoding/decoding data in accordance with the following encoding rules:

- Basic Encoding Rules (BER) and Distinguished Encoding Rules (DER) as published in the ITU-T X.690 and ISO/IEC 8825-1 standards.
- JSON Encoding Rules (JER) as published in the ITU-T X.697 and ISO/IEC 8825-8:2018 standards.

ASN1C for Python is capable of parsing all ASN.1 syntax as defined in the standards. It is capable of parsing advanced syntax including Information Object Specifications as defined in the ITU-T X.681 standard as well as Parameterized Types as defined in ITU-T X.683.

Chapter 2. ASN1C Command Line Interface (CLI)

Running ASN1C

The ASN1C compiler distribution contains command-line compiler executables as well as a graphical user interface (GUI) wizard that can aid in the specification of compiler options. Please refer to the *ASN1C C/C++ Compiler User's Manual* for instructions on how to run the compiler. The remaining sections describe options and configuration items specific to the Python version.

ASN1C Python Command Line Options

The following table shows a summary of the command line options that have meaning when Python code generation is selected:

Option	Argument	Description
-allow-ambig-tags		This option suppresses the check that is done for ambiguous tags within a SEQUENCE or SET type within a specification. Special code is generated for the decoder that assigns values to ambiguous elements within a SET in much the same way as would be done if the elements were declared to be in a SEQUENCE.
-asnstd	x680 x208 mixed	Parse ASN.1 syntax conforming to the specified standard. x680 (the default) refers to modern ASN.1 as specified in the ITU-T X.680-X.690 series of standards. x208 refers to the deprecated X.208 and X.209 standards. This syntax allowed the ANY construct as well as unnamed fields in SEQUENCE, SET, and CHOICE constructs. This option also allows for parsing and generation of code for ROSE OPERATION and ERROR macros and SNMP OBJECT-TYPE macros. The mixed option is used to specify a source file that contains modules with both X.208 and X.680 based syntax.
-ber	None	Generate functions that implement the Basic Encoding Rules (BER) as specified in the ASN.1 standards.
-compare	None	Generate a Python equality comparison method (*__eq__*) in each generated class.
-config	<filename>	This option is used to specify the name of a file containing configuration in-

Option	Argument	Description
		formation for the source file being parsed. A full discussion of the contents of a configuration file is provided in the *Compiler Configuration File* section.
-copy	None	Generate *copy_value* methods in all non-trivial classes to produce cloned deep-copy of an object instance.
-depends	None	Generate Python source files that contain only the productions in the main file being compiled and items those productions depend on from IMPORT files.
-der	None	Generate functions that implement the Distinguished Encoding Rules (DER) as specified in the ASN.1 standards.
-genbuild	[<filename>]	This option is used to generate a build script for invoking the compiler with the set of command-line options given in the command. If no file name is specified, the file will be named build.bat on Windows or build.sh on Linux/Mac.
-genPrint -print	None	Generate print methods to print object contents. For Python, this includes an *__str__* method to get a string representation of the object as well as a *print_value* to print the string representation to stdout.
-genTest -test	None	Generate test to populate a PDU object with random test data. This code will be embedded in the body of a generated writer program; thus the option only results in code being generated if -writer is also specified.
-I	<directory>	This option is used to specify a directory that the compiler will search for ASN.1 source files for IMPORT items. Multiple –I qualifiers can be used to specify multiple directories to search.
-json or -jer	None	Generate encode/decode functions that implement the Javascript Object Notation (JSON) Encoding Rules (JER) as specified in the X.697 ASN.1 standard.
-jer+	None	This option is used to generate encode/decode functions with ObjSys extensions to the Javascript Object

Option	Argument	Description
		Notation (JSON) Encoding Rules (JER) as specified in the X.697 ASN.1 standard. The chapter on JSON explains this option in detail.
-lax	None	Suppress generation of code to check constraints.
-list	None	Generate listing. This will dump the source code to the standard output device as it is parsed. This can be useful for finding parse errors.
-nodecode	None	Suppress generation of decode functions.
-noencode	None	Suppresses generation of encode functions.
-noIndefLen	None	Omit indefinite length tests in generated BER decode functions. These tests result in the generation of a fair amount of code. If you know that your application only uses definite length encoding, this option can result in a smaller code base size. Note that by definition it is enabled for DER code generations since these encoding rules do not use indefinite lengths.
-noOpenExt	None	Suppress the generation of open extension elements in constructs that contain extensibility markers. The purpose of the element is to collect any unknown items in a message. If an application does not care about these unknown items, it can use this option to reduce the size of the generated code and increase performance.
-o	<directory>	Specify the name of a directory to which all of the generated files will be written.
-pdu	<typeName>	Designate given type name to be a Protocol Definition Unit (PDU) type. By default, PDU types are determined to be types that are not referenced by any other types within a module. This option allows that behavior to be overridden. Note that for Python, the only effect this has is to cause this type to be used in generated reader/writer programs.
-reader	None	Generate a sample reader program to decode data.

Option	Argument	Description
-shortnames	None	Change the names generated by compiler for embedded types in constructed types.
-tables	None	Generate additional code for the handling of table constraints as defined in the X.682 standard.
-uniquenames	None	Automatically generate unique names to resolve name collisions in the generated code. Name collisions can occur, for example, if two modules are being compiled that contain a production with the same name. A unique name is generated by prepending the module name to one of the productions to form a name of the form <module>_<name>. Note that name collisions can also be manually resolved by using the typePrefix, enumPrefix, and valuePrefix configuration items (see the Compiler Configuration File section for more details)
-warnings	None	Output information on compiler generated warnings.
-writer	None	Generate a sample writer program to encode data.

Compiler Configuration File

In addition to command line options, a configuration file can be used to specify compiler options. These options can be applied not only globally but also to specificmodules and productions.

The basic structure of a configuration file is described in the C/C++ User's Guide. Configurations items that are applicable to Python code generation are described in the following sections.

Global Level

These attributes can be applied at the global level by including them within the `<asn1config>` section:

Name	Values	Description
<includedir></includedir>	<Include directory>	This configuration item is used to specify a directory that will be searched for IMPORT files. It is equivalent to the -I command-line option.

Module Level

These attributes can be applied at the module level by including them within a <module> section:

Name	Values	Description
<name> </name>	module name	This attribute identifies the module to which this section applies. Either this or the <oid> element/attribute is required.
<oid>	module OID (object identifier)	This attribute provides for an alternate form of module identification for the case when module name is not unique. For example, a given ASN.1 module may have multiple versions. A unique version of the module can be identified using the OID value.
<include types="names" values="names"/>	ASN.1 type or values names are specified as an attribute list	This item allows a list of ASN.1 types and/or values to be included in the generated code. By default, the compiler generates code for all types and values within a specification. This allows the user to reduce the size of the generated code base by selecting only a subset of the types/values in a specification for compilation. Note that if a type or value is included that has dependent types or values (for example, the element types in a SEQUENCE, SET, or CHOICE), all of the dependent types will be automatically included as well.
<exclude types="names" values="names"/>	ASN.1 type or values names are specified as an attribute list	This item allows a list of ASN.1 types and/or values to be excluded from the generated code. By default, the compiler generates code for all types and values within a specification. This is generally not as useful as the *include* directive because most types in a specification are referenced by other types. If an attempt is made to exclude a type or value referenced by another item, the directive will be ignored.
<sourceFile> </sourceFile>	source file name	Indicates the given module is contained within the given ASN.1 source file. This is used on IMPORTs to instruct the compiler where to look for imported definitions.
<valuePrefix> </valuePrefix>	prefix text	This is used to specify a prefix that will be applied to all generated value constants within a module. This can be used to prevent name clashes if multiple modules are involved that use a common name for two or more different value declarations.

Production Level

These attributes can be applied at the production level by including them within a <production> section:

Name	Values	Description
<name> </name>	production name	This attribute identifies the production (type) to which this section applies. The name may also be specified as an attribute. In either case, it is required.
<isTBCDString/>	n/a	This item is used to indicate that this production is to be encoded and decoded as a telephony binary coded string (TBCD). This is type is not part of the ASN.1 standards but is a widely used encoding format in telephony applications.
<typePrefix> </typePrefix>	prefix text	This is used to specify a prefix that will be applied to all generated Python class names. This can be used to prevent name clashes if multiple modules are involved in a compilation and they all contain common names.

Element Level

These attributes can be applied at the element level by including them within an <element> section:

Name	Values	Description
<name> </name>	element name	This element identifies the element within a SEQUENCE, SET, or CHOICE construct to which this section applies. It may also be specified as an attribute. In either case, it is required.
<isOpenType/>	n/a	This flag variable specifies that this element will be decoded as an open type (i.e. skipped). Refer to the section on deferred decoding and partial decoding for further information. Note that this variable can only be used with BER, CER, or DER encoding rules.
<notUsed/>	n/a	This flag variable specifies that this element will not be used at all in the generated code. It can only be applied to optional elements within a SEQUENCE or SET, or to elements within a CHOICE. Its purpose is for production of more compact code by allowing users to configure out items that are of no interest to them.
<skip/>	n/a	This flag variable specifies that this element will be skipped during decod-

Name	Values	Description
		ing. Refer to the section on deferred decoding and partial decoding for further information. Note that this variable can only be used with BER, CER, or DER encoding rules.

Compiler Error Reporting

Errors that can occur when generating source code from an ASN.1 source specification take two forms: syntax errors and semantics errors.

Syntax errors are errors in the ASN.1 source specification itself. These occur when the rules specified in the ASN.1 grammar are not followed. ASN1C will flag these types of errors with the error message 'Syntax Error' and abort compilation on the source file. The offending line number will be provided. The user can re-run the compilation with the '-l' flag specified to see the lines listed as they are parsed. This can be quite helpful in tracking down a syntax error.

The most common types of syntax errors are as follows:

- Invalid case on identifiers: module name must begin with an uppercase letter, productions (types) must begin with an uppercase letter, and element names within constructors (SEQUENCE, SET, CHOICE) must begin with lowercase letters.
- Elements within constructors not properly delimited with commas: either a comma is omitted at the end of an element declaration, or an extra comma is added at the end of an element declaration before the closing brace.
- Invalid special characters: only letters, numbers, and the hyphen (-) character are allowed. Programmers tend to like to use the underscore character (_) in identifiers. This is not allowed in ASN.1. Conversely, Python does not allow hyphens in identifiers. To get around this problem, ASN1C converts all hyphens in an ASN.1 specification to underscore characters in the generated code.

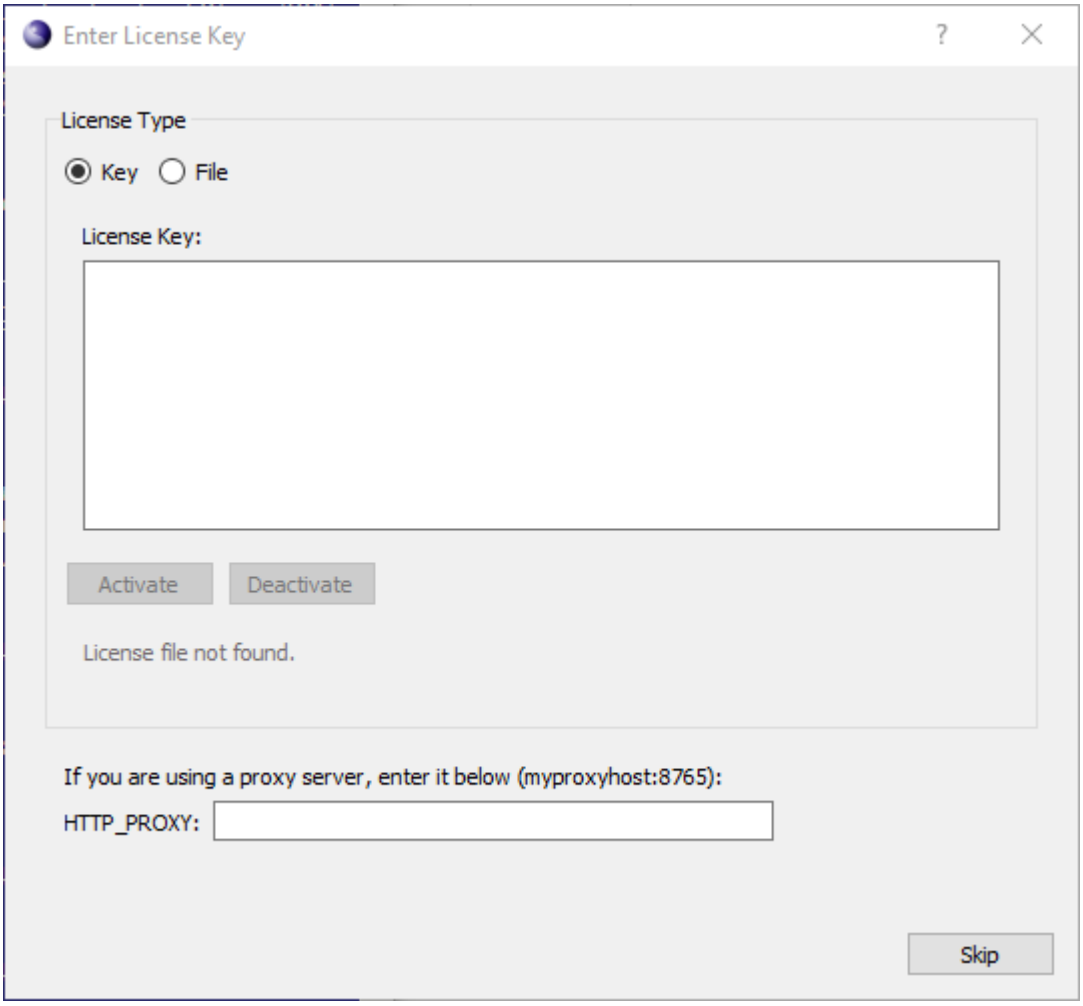
Semantics errors occur on the compiler back-end as the code is being generated. In this case, parsing was successful, but the compiler does not know how to generate the code. These errors are flagged by embedding error messages directly in the generated code. The error messages always begin with an identifier with the prefix '**%ASN-**',. A search can be done for this string in order to find the locations of the errors. A single error message is output to stderr after compilation on the unit is complete to indicate error conditions exist.

Chapter 3. ASN1C GUI Users Guide

Quick Start

In this section, we will demonstrate running ACGUI, creating a new ASN.1 schema, and compiling it to C for BER data. The process is similar for other languages.

First, start ACGUI. You may see the window below asking you to activate a license key.



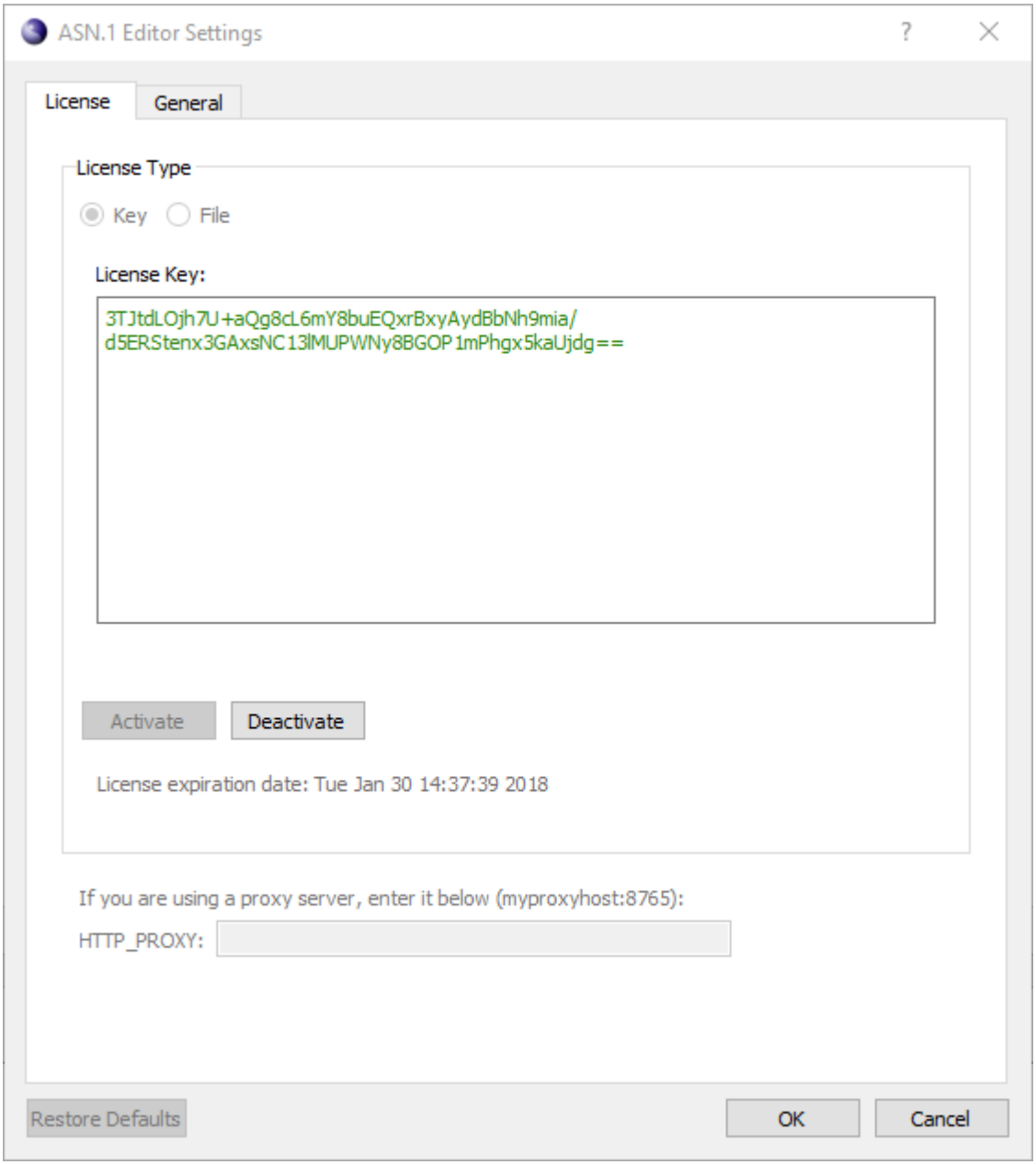
If you see this window, and it's not showing a current license key, you can right click in the text box and paste in your license key. Then click the Activate button to unlock ASN1C.

In some cases, however, the window will appear and show a current license key. In these cases the key being shown is probably expired. You need to deactivate the current key first by clicking the Deactivate button. Then you can right click in the text box and paste in your current license key. Then click the Activate button to unlock ASN1C.

If you have an osyslic.txt license file in a location where the GUI doesn't look, you can click the Import button to find that file and use it to unlock ASN1C instead of activating a key.

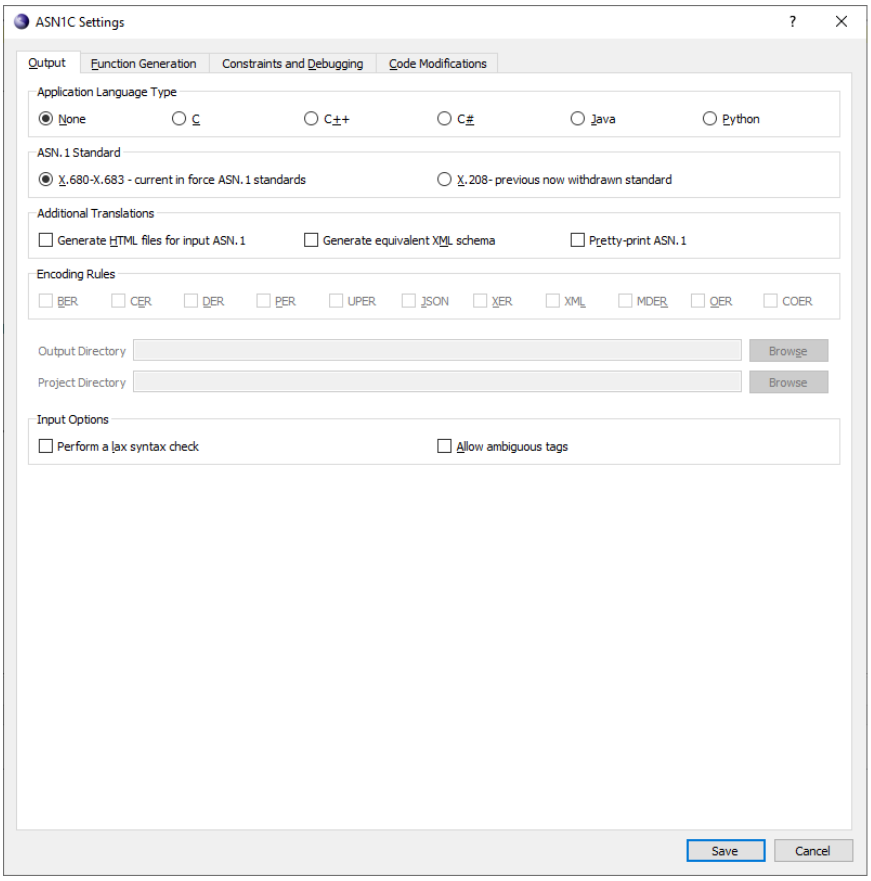
If you click the Skip button, you can expore the features of the GUI, but you will not be able to generate any code.

Should you ever need to activate a different license key even though you have a current one (for example, you purchase ASN1C before your evaluation key expires), you should deactivate the existing license first. This deactivation can be done from the GUI by choosing Tools...Options and then clicking on the License tab. You will see a window similar to this one:



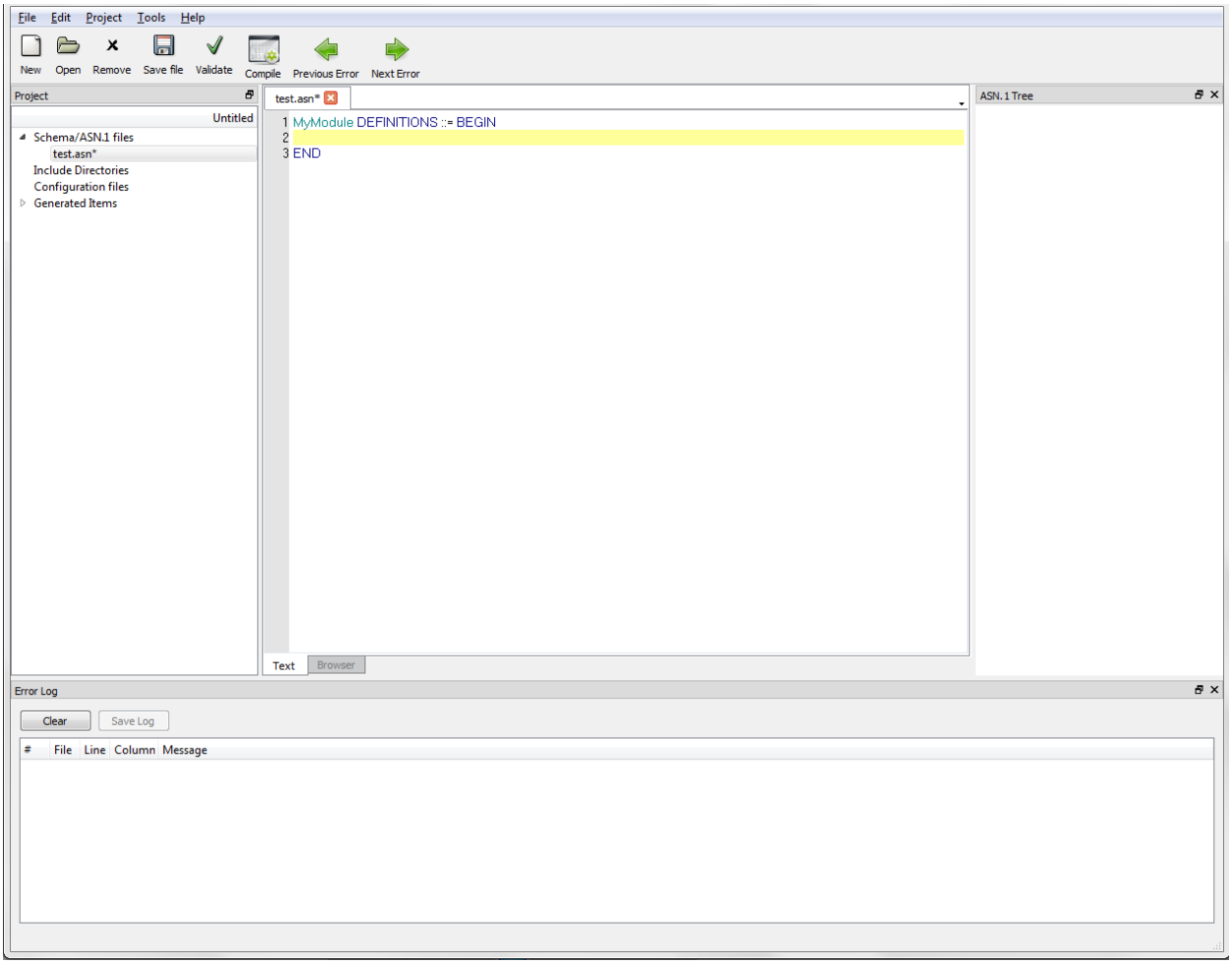
Click the Deactivate button to deactivate the existing license. You can then do the same sequence to bring up the window again and activate your new key.

Once the program is running, we'll create a new project to store all of our settings. To do this, select "Project->New Project..." from the menus.



In the first tab, "Output", under "Application Language Type", select "C". Below that, under "Encoding Rules", select "BER". Finally, choose an output directory for the generated files. Once the settings are correct, click "Save". These settings can be changed at any time by selecting "Project->Project Settings..." from the menus.

Next, we'll create a new schema file for our project. Click the "New" button in the toolbar or select "File->New Schema File" from the menus. A dialog box will be shown to provide a name for the new file. Once entered, the file will be added to the project window under "Schema/ASN.1 files" and its empty contents will be shown in the editor.

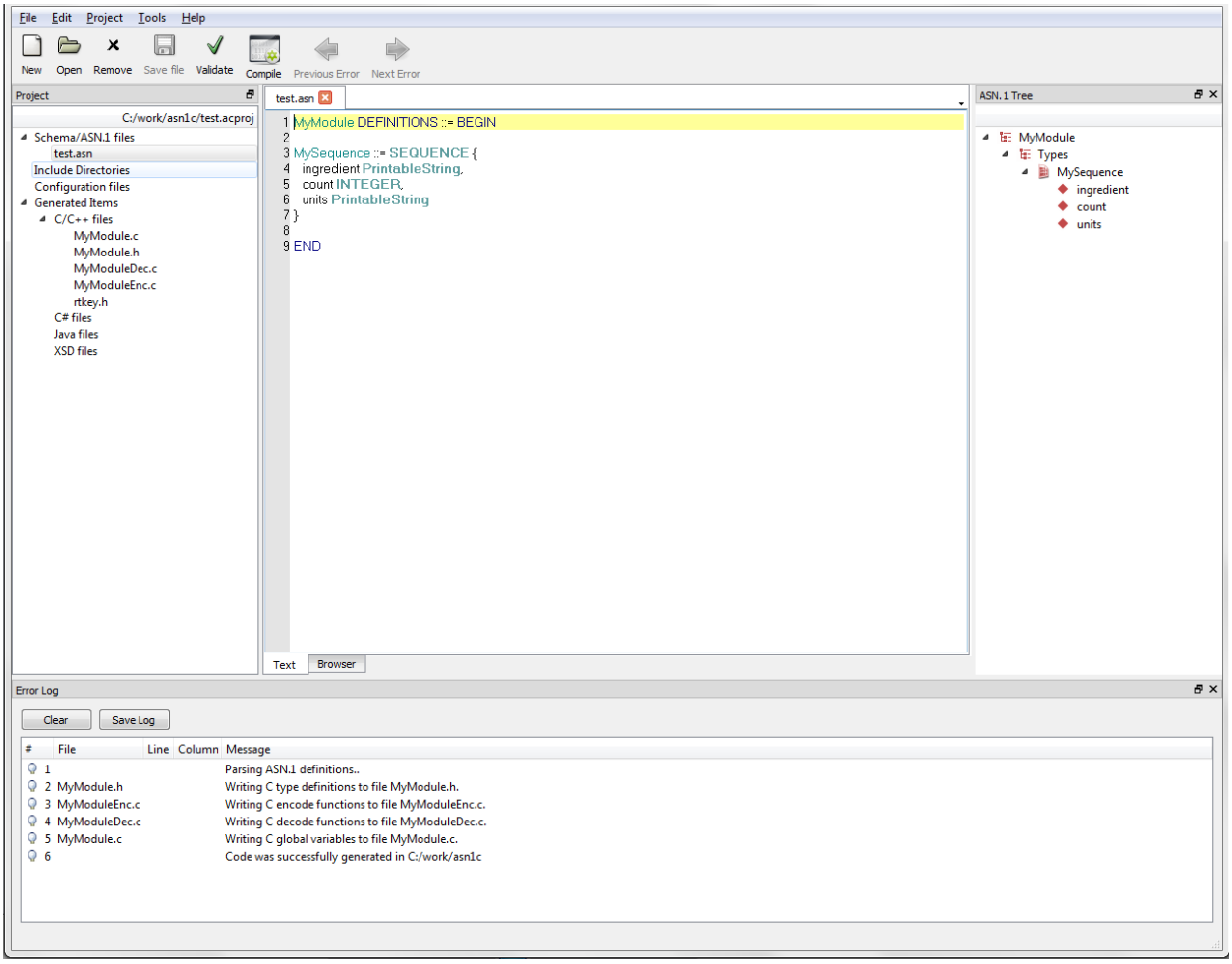


Now we need to write our schema between the "DEFINITIONS ::= BEGIN" and "END" statements in the file. Enter something like:

```
MySequence ::= SEQUENCE {
    ingredient PrintableString,
    count INTEGER,
    units PrintableString
}
```

When you are satisfied with the schema you've created, click the "Validate" button to make sure there are no errors. Since the new schema file hasn't been saved yet, ACGUI will ask if it should be. Save the new file and ACGUI will then validate it. If the schema has errors, the log at the bottom of the window will show them. Otherwise, it is safe to move on.

Since we've already set up our project, we can click the "Compile" button to generate code according to our project settings. If all goes well, the project window will show a list of generated files under "Generated Items" in the section for the selected language. If there were any errors, they will be shown in the log.



At this point, project settings can be changed and schema files can be edited as needed.

Creating a Project

Since there are a large number of options available in the code generation process, ACGUI allows settings to be saved in project files for reuse. Project files can be created, opened, and saved from the "Project" menu. If no project file is explicitly used, a dummy project will be implicitly created and can be saved to a file at a later time.

Project assets, such as ASN.1 schemas and generated source files, are visible in the "Project" window. Project settings can be changed via the Project Settings window, accessible by selecting "Project->Project Settings..." from the menubar.

Editing Schemas

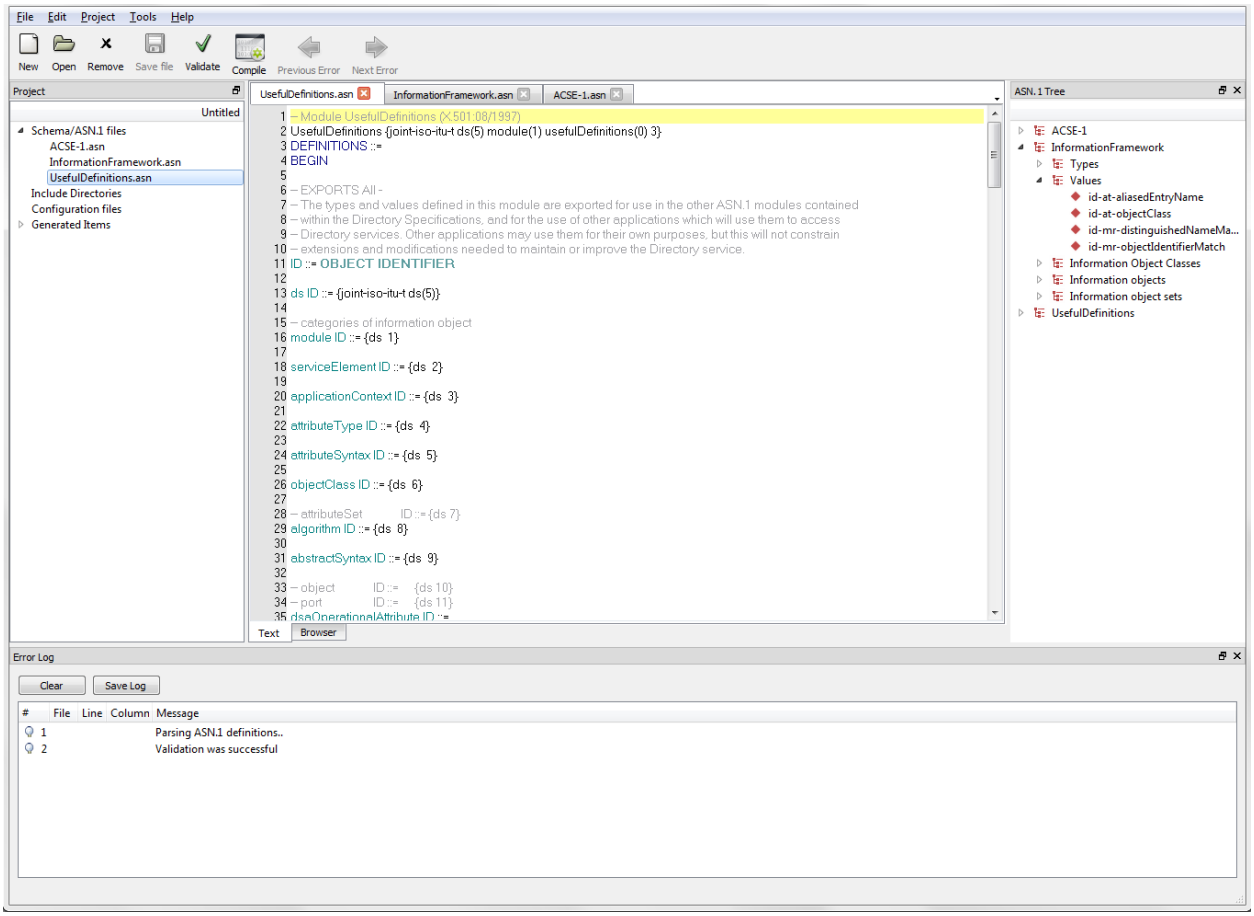
The central area of the ACGUI window is dedicated to editing ASN.1 schema definition files. To create a new file, click the "New" button in the toolbar or select "File->New Schema File" in the menus. To open an existing schema file, either click the "Open" button in the toolbar or select "File->Open File..." from the menus. In both cases, the file will be added to the project and the editor will show it. Clicking on an ASN.1 schema file name in the project window will also display that file in a tab in the editor.

At any point during editing, the schema can be saved and checked for proper syntax by clicking the "Validate" button in the toolbar.

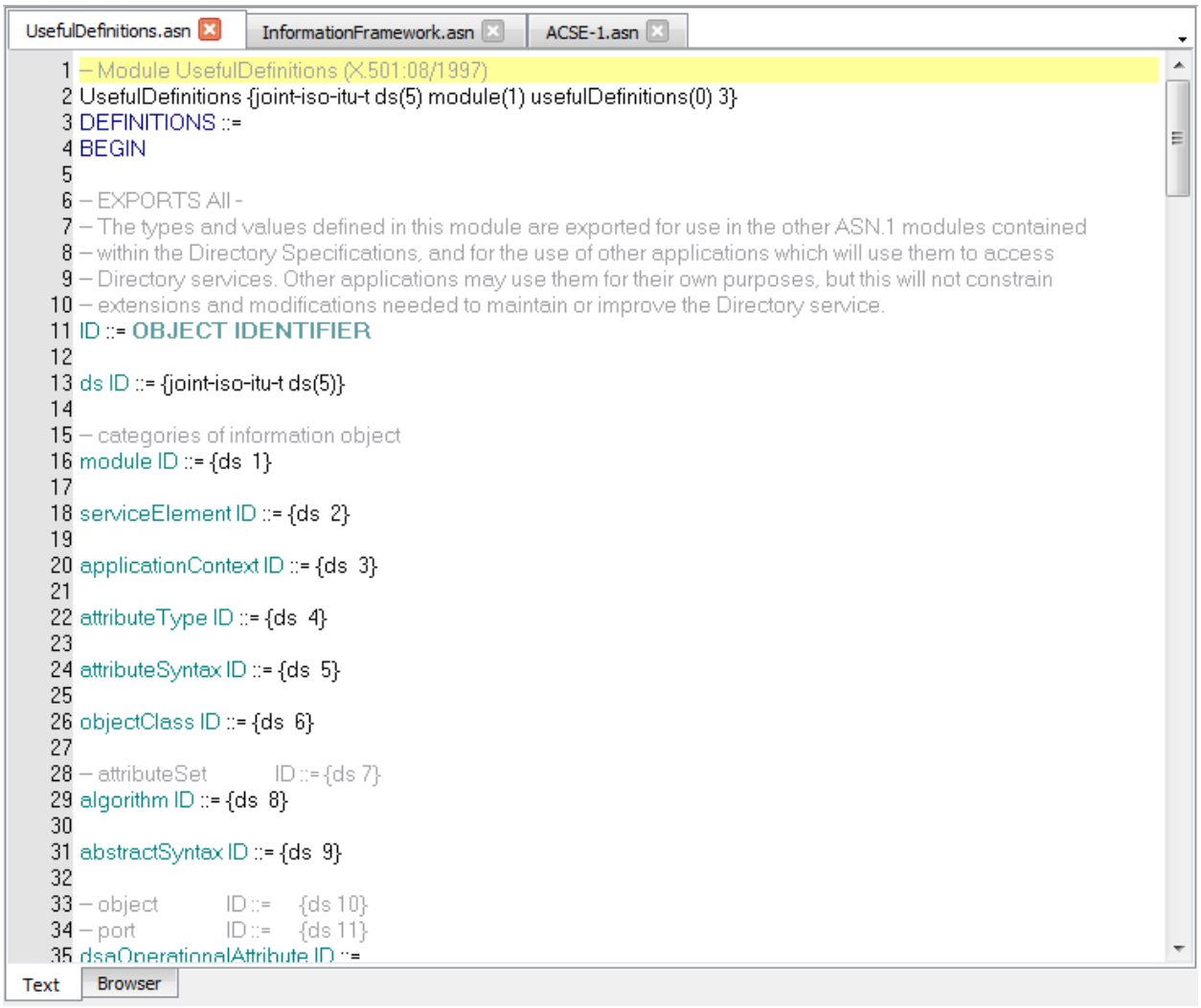
Compiling

Once a project has been created and schemas added, the schemas may be compiled. This assumes that a target language has been selected in the project. Click the "Compile" button or select "Tools->Compile" from the menus. If any files have unsaved changed, a dialog box will be displayed prompting the user to save them. Once saved, the compiler will run, generating source code and related files. From here, changes can continue to be made to the schema and to project settings and recompilation done as needed.

Interface



Editor



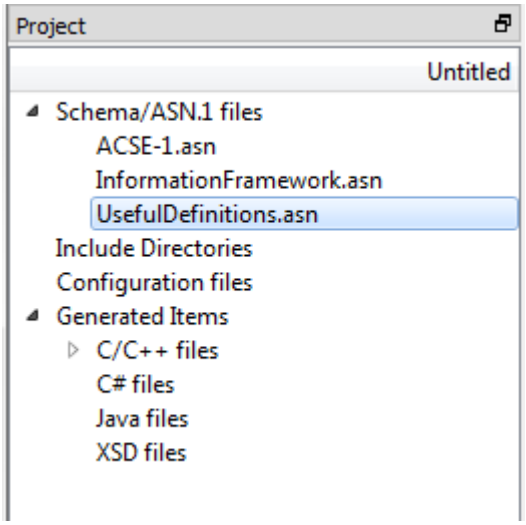
The central part of the ACGUI window is the schema editor. From here, schema files can be viewed and edited. To begin editing an ASN.1 schema, create a new file or open an existing file via the toolbar or menu. The file will be added to the current project and shown in the editor.

The editor window is also used to display a schema browser for navigating a validated schema. To display the browser after validating a schema, click on an item in the ASN.1 Tree window. The browser will display a hyperlinked version of the schema, centered on the definition of the selected item. Clicking the names of other defined types in the browser will cause their definitions to be shown.

By default, documents are displayed in tabs in the editor. Tabs "Text" and "Browser" at the bottom of the window are for schema editing and hyperlinked schema browsing, respectively. At the top of the "Text" tab, each schema file currently being edited has a tab.

Alternatively, ACGUI can display documents in separate subwindows. To change this, select "Tools->Options..." from the menus. In the "General" tab of the options window, change "Open files" from "In tabs" to "In MDI windows". Click "OK" and restart ACGUI. Now, open files will be displayed as separate windows within the main ACGUI window. This option is useful for viewing two files simultaneously, for example.

Project Window

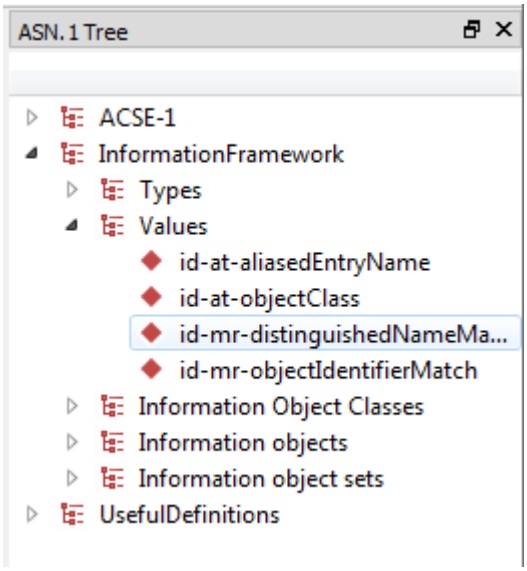


The project window allows the user to interact with project assets.

Schema/ASN.1 files	Files containing the current project's ASN.1 schema definitions.
Include directories	Directories containing auxiliary ASN.1 schema files. The current project's schema may import definitions from modules defined in an included directory.
Configuration file	An ASN1C compiler configuration file.
Generated Items	A listing of the files generated by the compiler, separated by target language.

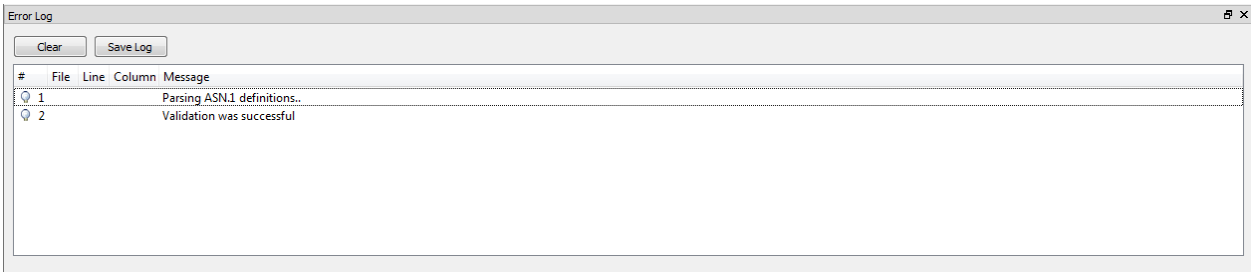
Clicking on a schema file or configuration file in the project window will open that file in the editor. A right-click context menu is also provided for schema files, include directories, and configuration file for adding or removing these assets from the project.

ASN.1 Tree Window



Once a schema has been validated or compiled in ACGUI, the ASN.1 Tree window provides an interactive view of the ASN.1 types defined in it. At the top level of the tree, the modules of the schema are shown. Each of these modules can be expanded to reveal branches for the types, values, information objects, etc. defined in each of them. Clicking on any node of the tree will show the relevant ASN.1 definition in a built-in browser in the editor window.

Error Log Window



The Error Log window displays messages related to schema validation and compilation. Whenever a schema is successfully validated or compiled, the Error Log will report a success. If an error occurs, an error message will be displayed.

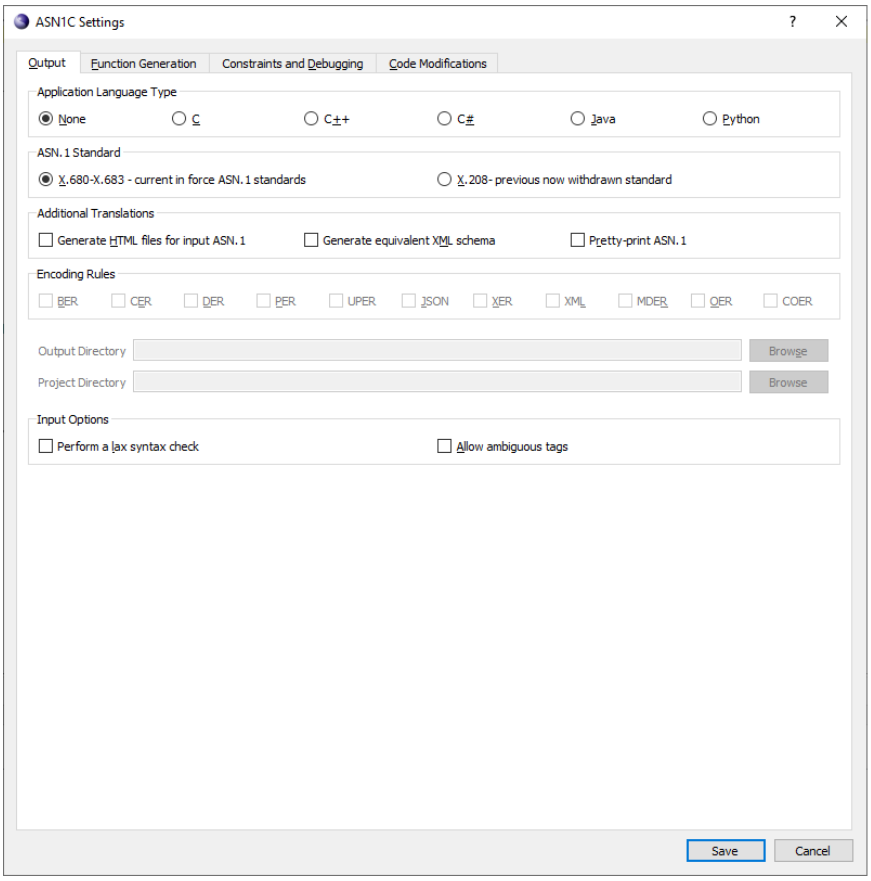
In many cases, an error will be associated with a particular portion of the schema being compiled. In these cases, clicking on an error will open the schema editor to the location that the error occurred. If more than one error is reported, clicking the "Next Error" and "Previous Error" buttons in the toolbar will move the editor window to the part of the schema where the next or previous error occurred.

When the reported errors are no longer needed, they can be cleared by clicking the "Clear" button in the Error Log window.

Project Settings

The project settings window is where details such as encoding rules, target language, and code features to generate are modified.

Output tab



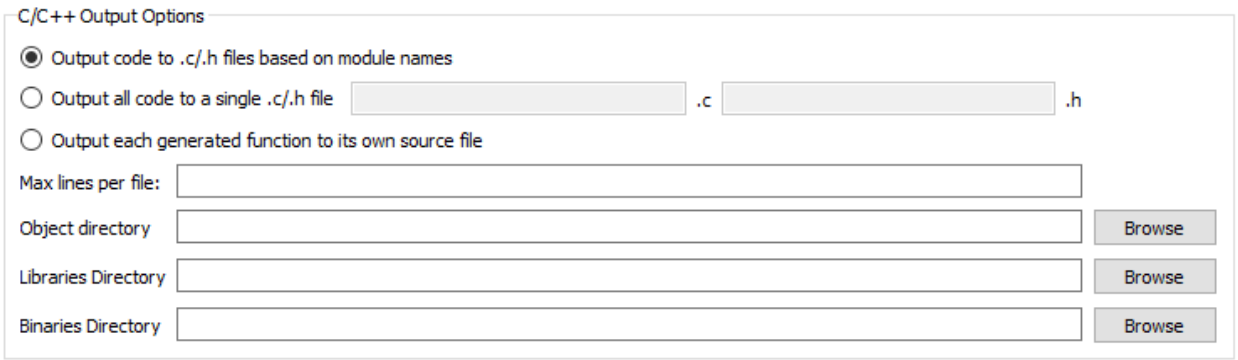
The "Output" tab contains options for selecting a target language, encoding rules, output directory. In order to compile a schema, a target language must be selected under "Application Language Type".

"Additional Translations" contains several options for generating transformed versions of the input schema, such as HTML or pretty-printed.

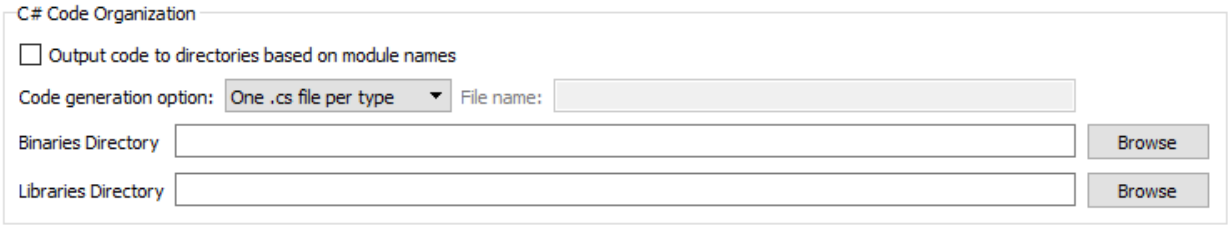
"Encoding Rules" allows for one or more encoding rule set to be selected for generated code.

"Input Options" affect how strict the compiler is when parsing the ASN.1 schema.

Depending on the target language selected, additional options are shown.



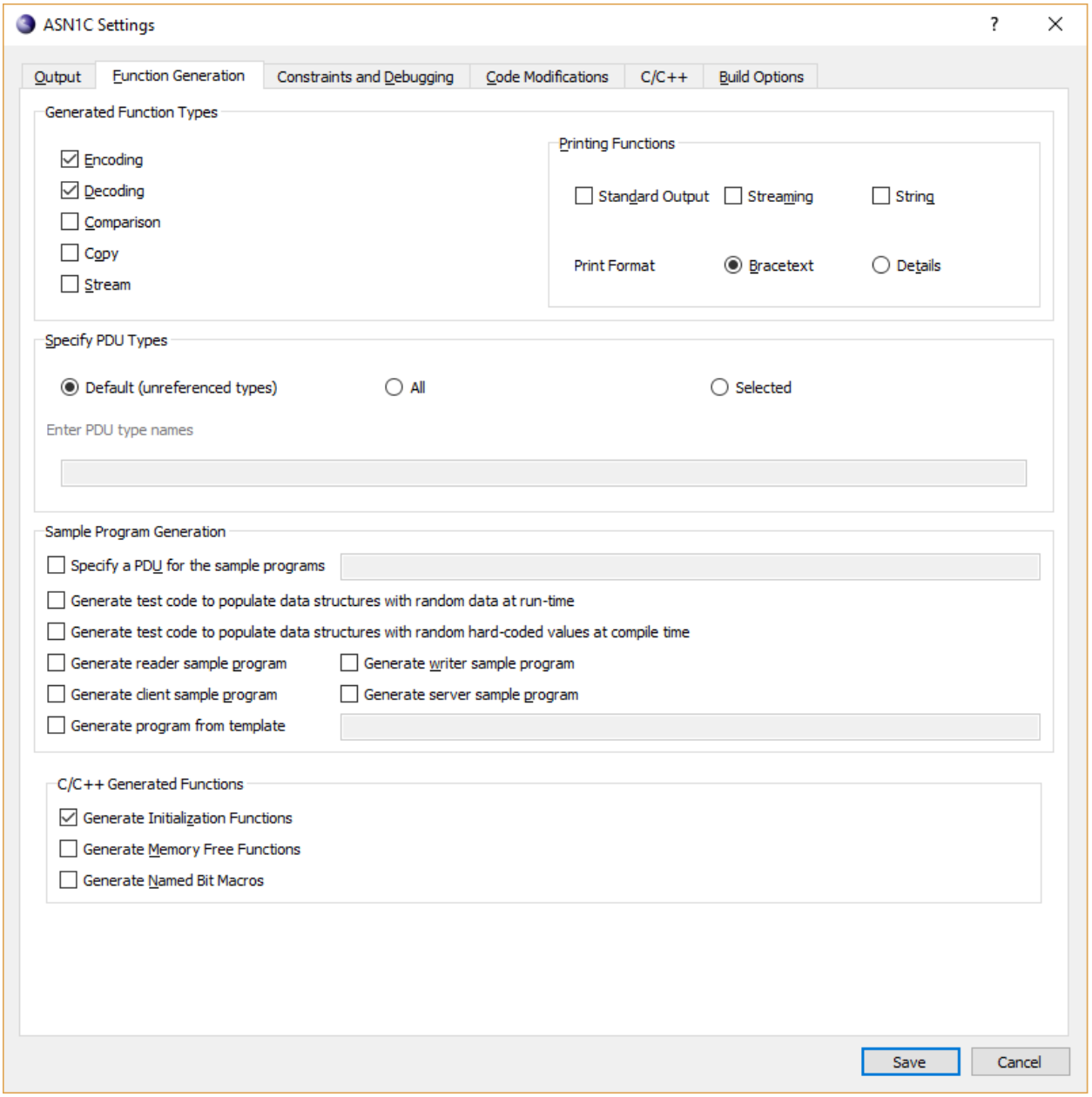
For C or C++ target languages, "C/C++ Output Options" controls how generated code is distributed across source files.



For C#, "C# Code Organization" controls how generated code is distributed across source files and how files are organized into directories.

For Java, "Java Code Organization" allows generated code to be organized into directories based on the ASN.1 module for which they were generated. Alternatively, generated files will be placed directly into the output directory.

Function Generation tab

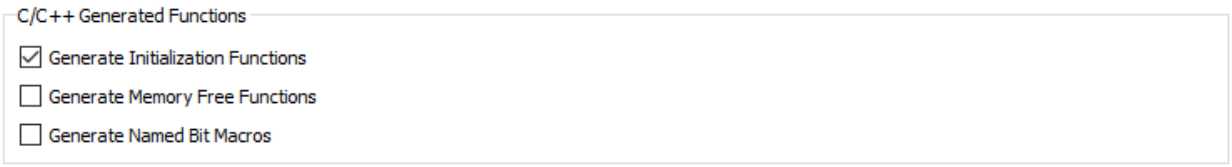


The "Function Generation" tab provides settings for what functionality to include in generated code. Options under "Generated Function Types" provide granular control of what functions to generate. The printing functions allow for various printing schemes to be generated, such as print-to-string and print-to-standard-output, and how the printed data should be formatted.

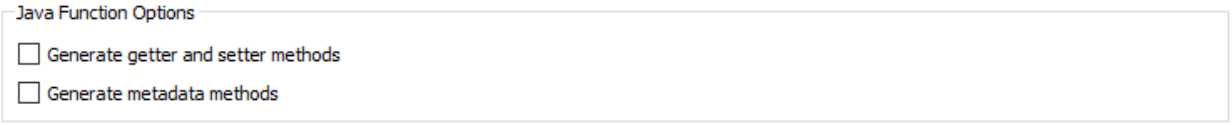
The "Specify PDU Types" area provides options for telling ASN1C what productions to choose as PDUs.

"Sample Program Generation" allows simple encoding and decoding programs, which demonstrate using the generated code, to be generated. The sample writer program can optionally encode randomly-generated test data.

Depending on the target language selected, additional options may be shown here.

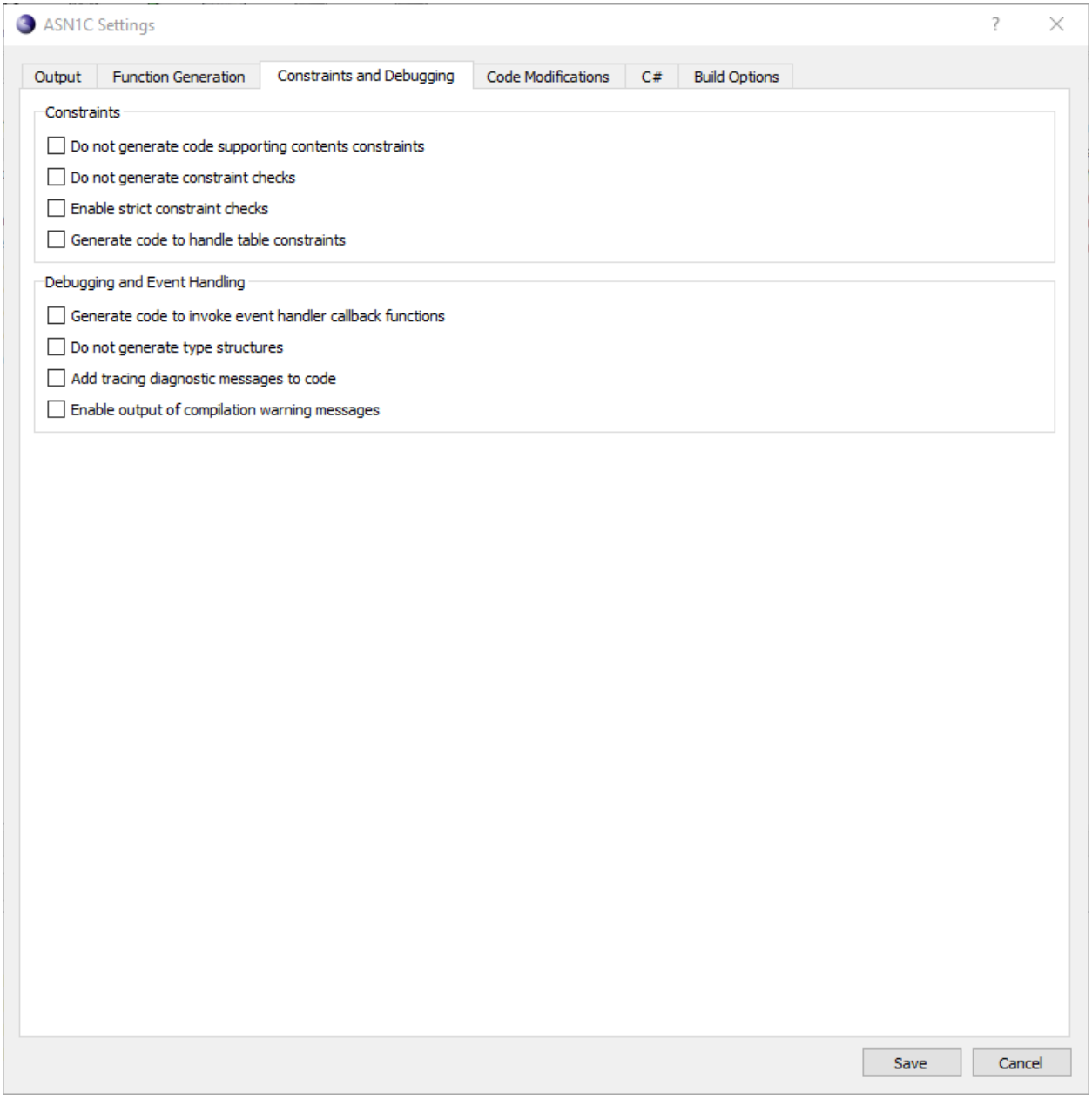


For C and C++, additional functions for memory management and macros for dealing with named bits in BIT STRINGs can be generated. Initialization functions are generated by default. They may be turned off in the window.



For Java, get and set methods can be generated for members of generated classes. It is also possible to generate methods that can fetch certain types of metadata (for example, if an element is optional). A similar option exists for C#.

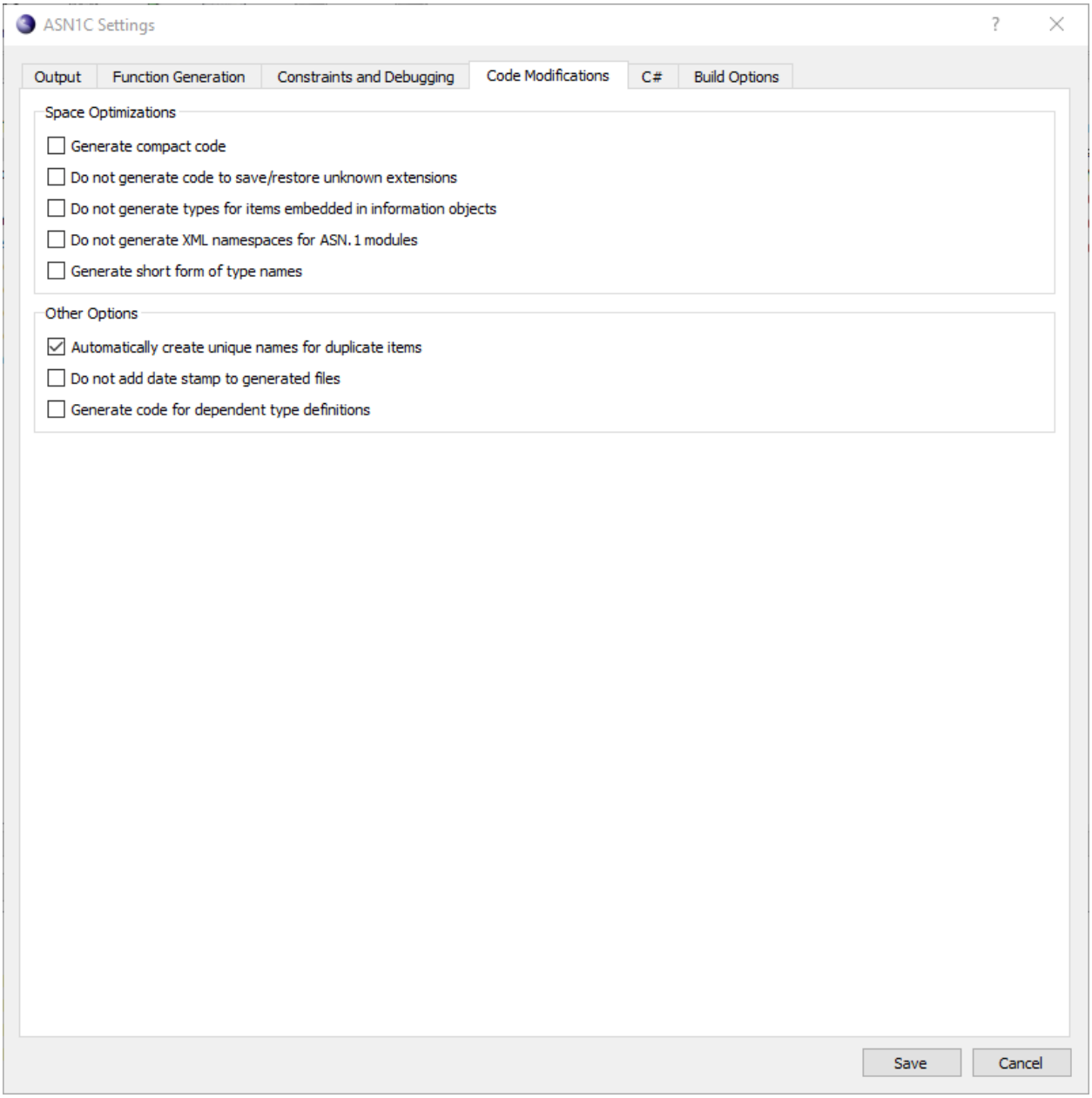
Constraints and Debugging tab



The "Constraints and Debugging" tab holds settings related to constraint handling, event handling, and logging in generated code. Under "Constraints", various types of constraint checks can be added or removed from generated code.

In "Debugging and Event Handling", settings for adding debug tracing and event hooks are available. In addition to enabling event callbacks, generation of type structures can also be disabled, in which case generated decode functionality will simply call user-created event handlers and not perform its own decoding operation.

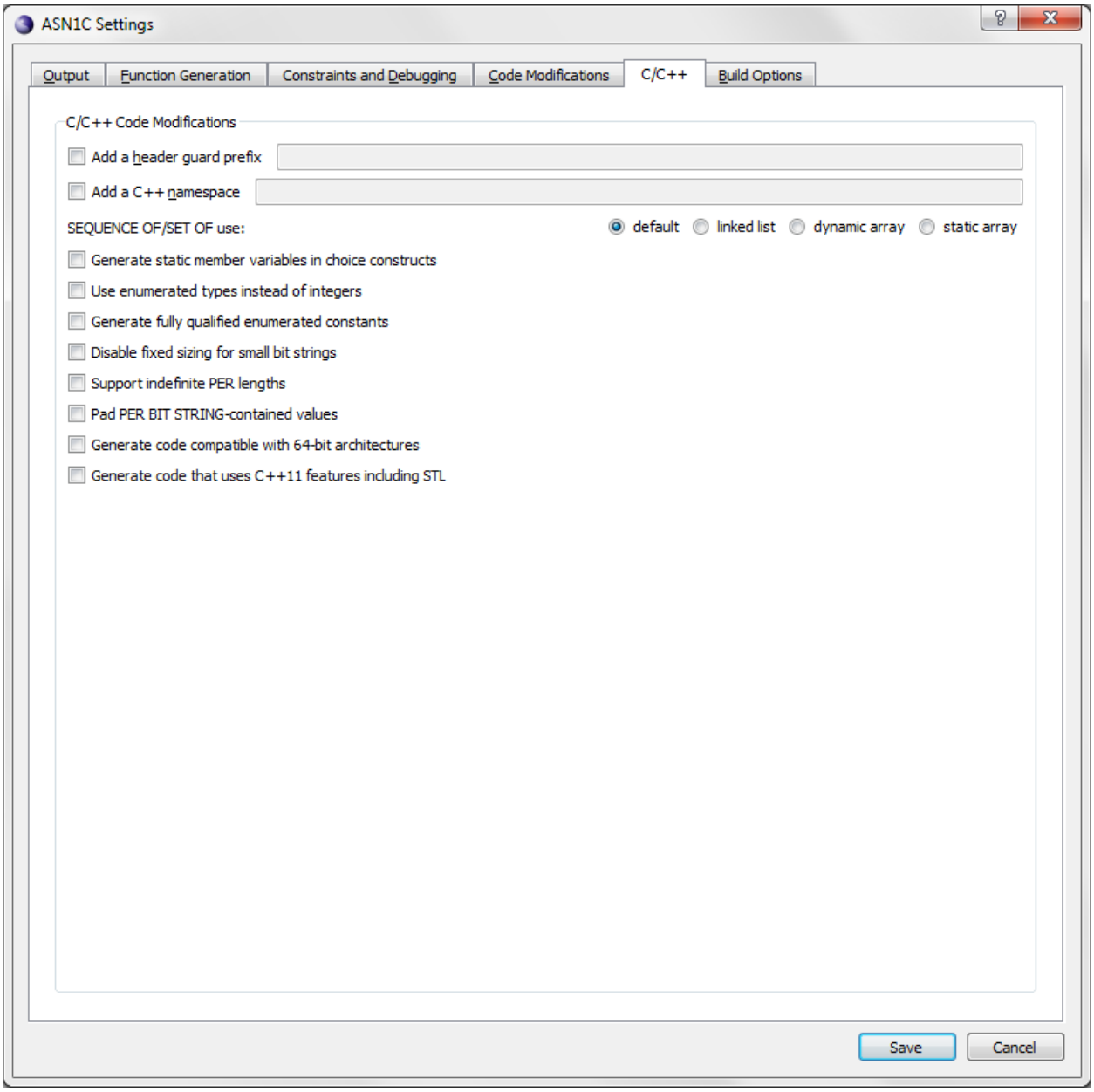
Code Modifications tab



Under the "Code Modifications" tab are a number of options for generating simplified code. In "Space Optimizations", these mainly regard the removal of unwanted or unneeded functionality and shortening names of generated types.

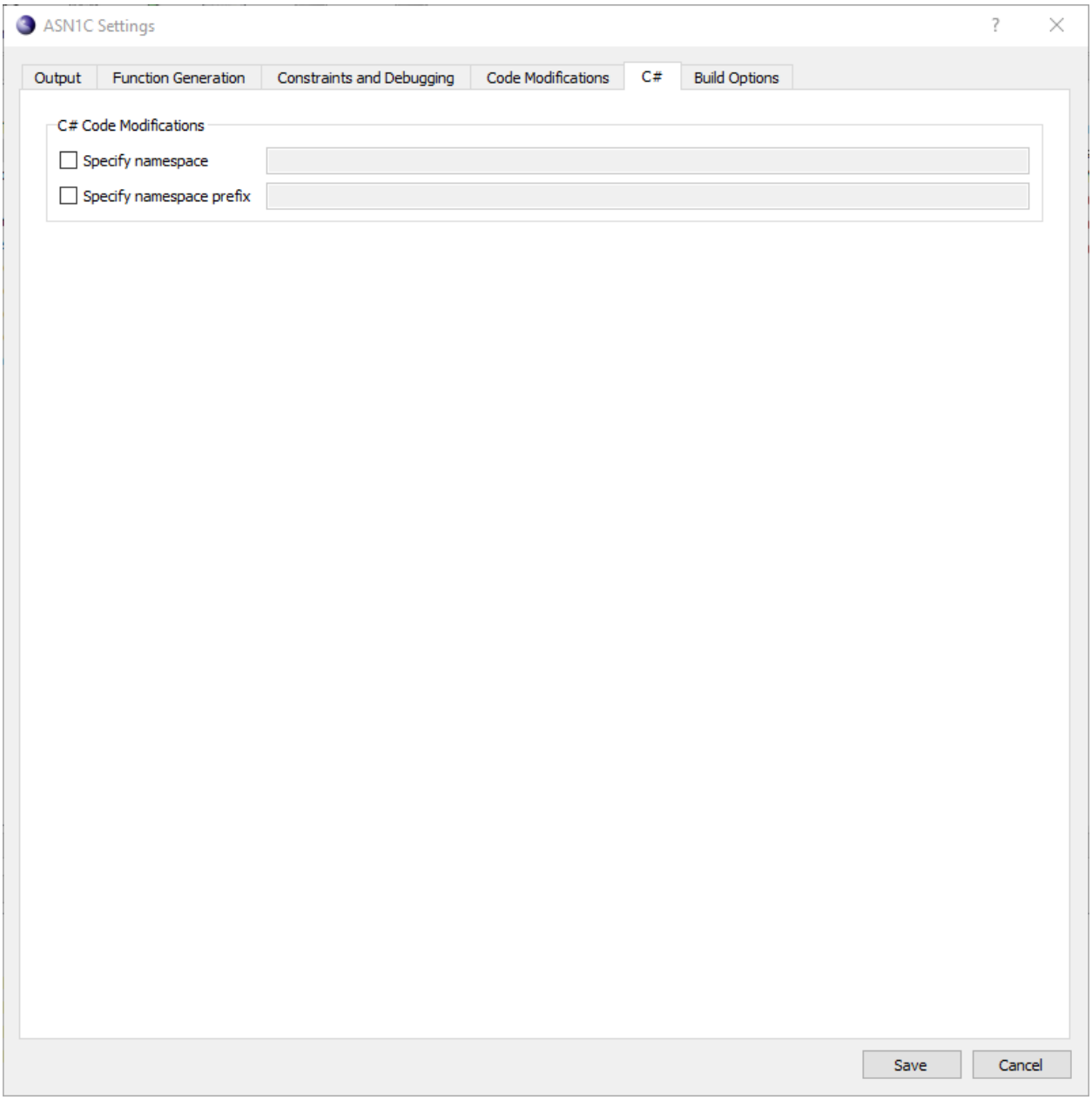
"Other Options" provide several miscellaneous settings, including the option of generating code for types that have been imported into the current schema.

Additional code modification options that are language-specific are shown in a separate tab next to the "Code Modifications" tab. The lable on this tab will change based on the language selected. For C/C++, it is as follows:



In this case, code modifications include several settings for adjusting how ASN.1 types are mapped to native C/C++ types.

For C#, the tab is as follows:

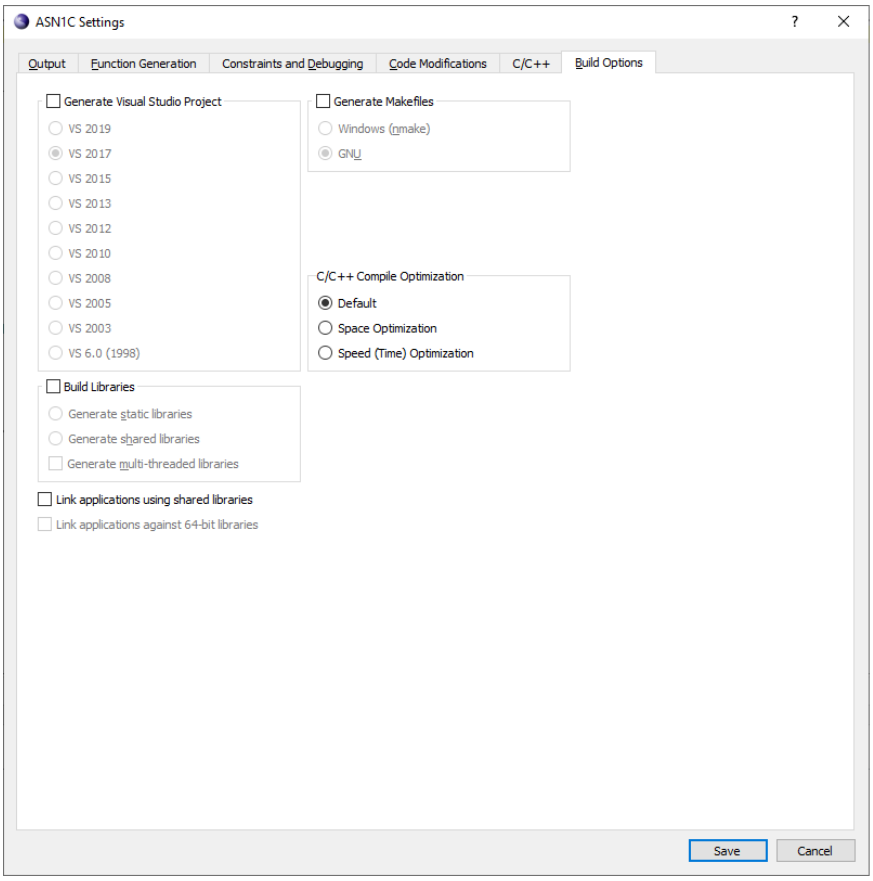


In this, modifications allow for manipulating the namespace into which code is generated. The Java tab contains similar options.

Build Options tab

When a target language other than "None" is selected, an additional "Build Options" tab contains language environment-specific settings for generating makefiles and build scripts.

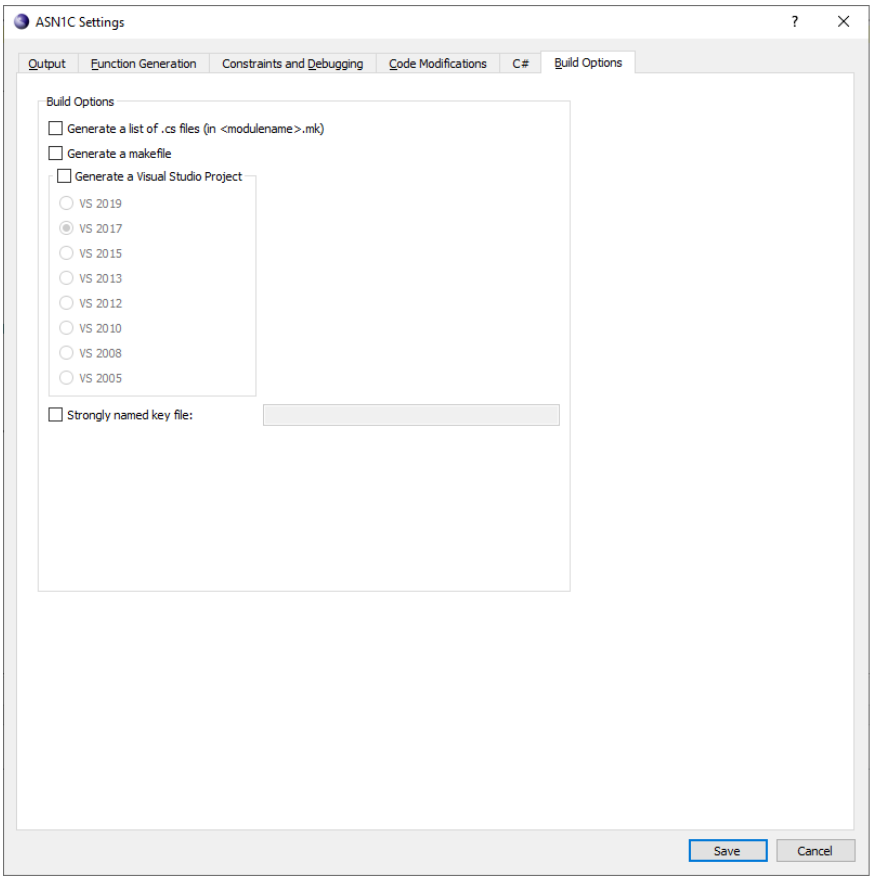
For C or C++, the window is as follows:



A makefile can be generated in either Windows or GNU format. For Windows, a Visual Studio project can also be generated. Under "Build Libraries", which will generate the build script to build a library rather than an executable, the desired variety of library can be selected.

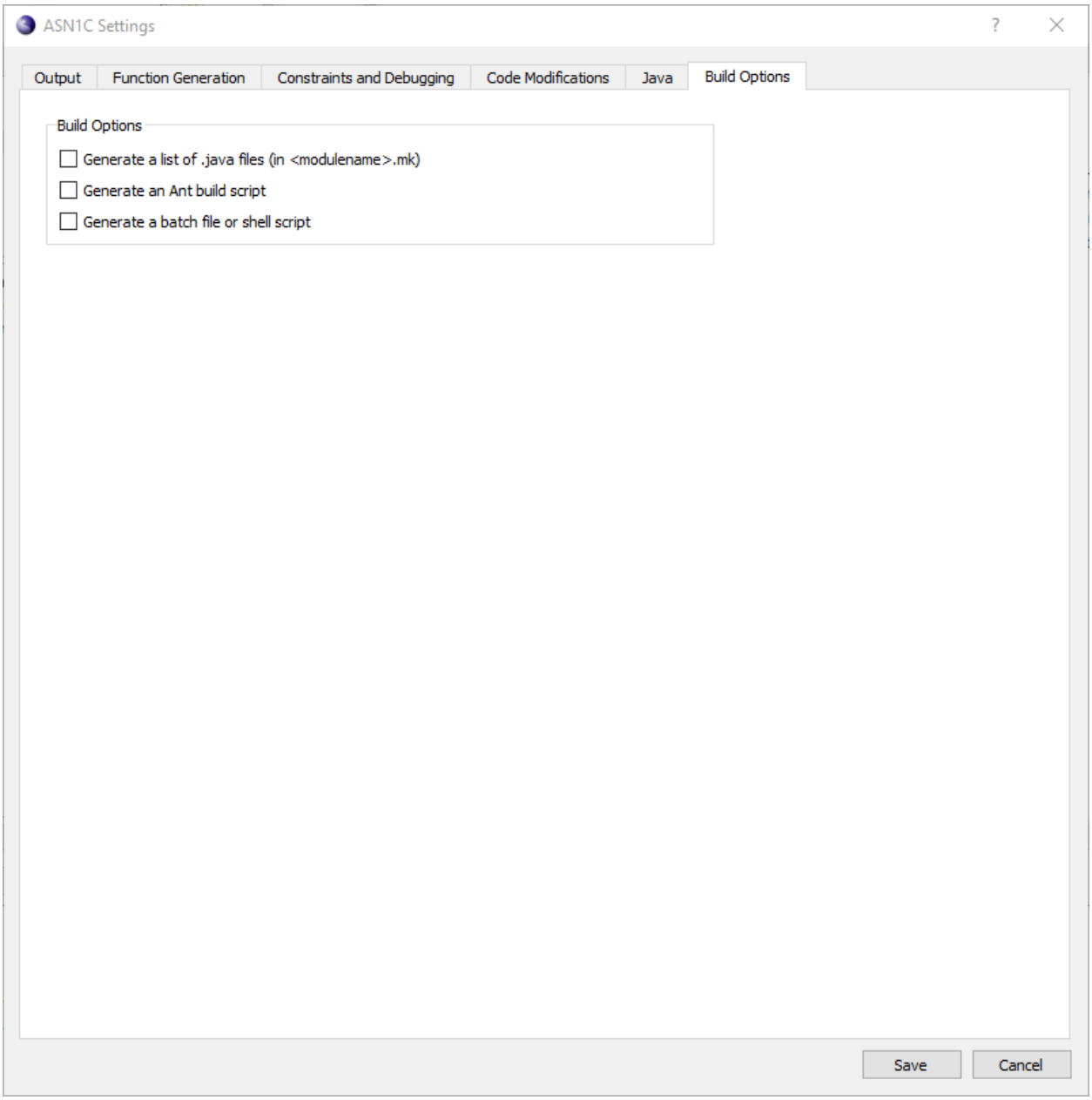
"C/C++ Compile Optimization" allows for setting whether Space or Time optimization qualifiers should be added to the C compilation command-line in the makefile.

For C#, the following options are displayed:



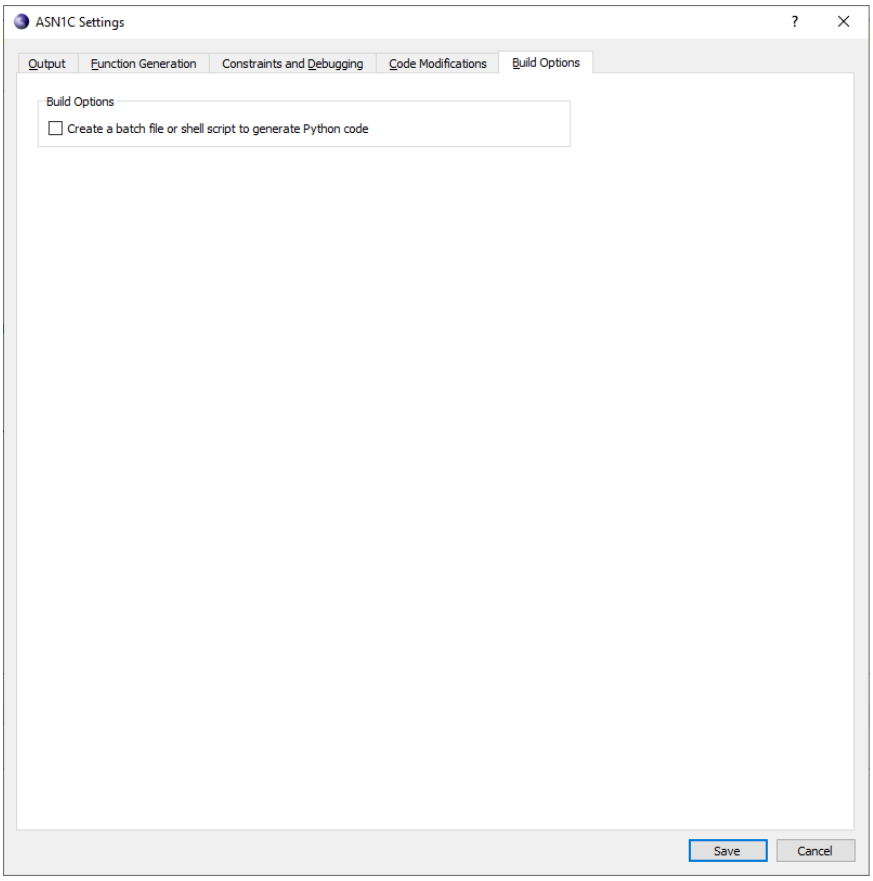
Similarly for C#, a makefile or Visual Studio project can be created, optionally including a *.mk file listing the files generated. An option to specify a strongly named key file also exists.

For Java, the following is displayed:



Java can also provide a *.mk generated file list, as well as Ant build script and a batch or shell script.

For Python, the following is displayed:



For Python ASN1C can create a batch file (Windows) or shell script (non-Windows) that will generate the Python code as set up by the GUI settings.

Chapter 4. Generated Python Source Code

A Python source file with extension '.py' may be generated for each module in an ASN.1 specification source file. If the module does not contain any ASN.1 types that would result in the mapping to a Python class or value definition, no Python file is generated.

General Form of a Generated Python Source File

The following items may be present in a generated source file:

- Import statements
- Simple value definitions
- Class declarations
- Complex value definitions

Import Statements

Import statements are generated to include definitions from the built-in run-time components as well as from other ASN.1 modules from which definitions were imported. As example of generated imports is as follows:

```
import osyspyrt.asn1error as exc
import osyspyrt.asn1univtype as univ
import osyspyrt.asn1ber as ber
```

Imports from the ASN.1 common run-time libraries are first. These are prefixed by the package name "osyspyrt". The symbols are mapped to a prefix to prevent name clashes with items in imported ASN.1 specifications or the specification being compiled. These are followed by import statements corresponding to ASN.1 IMPORT statements in the module being compiled.

Simple Value Definitions

Simple value definition are primitive typed values that do not have any dependencies on generated classes. Values of type BOOLEAN, INTEGER, NULL, OBJECT IDENTIFIER, character string, or binary string fall into this category. Global assignments are generated for these types. An example would be as follows:

```
id_sha256 = [2, 16, 840, 1, 101, 3, 4, 2, 1]
id_sha384 = [2, 16, 840, 1, 101, 3, 4, 2, 2]
id_sha512 = [2, 16, 840, 1, 101, 3, 4, 2, 3]
pkcs_1 = [1, 2, 840, 113549, 1, 1]
```

These are OBJECT IDENTIFIER values from the PKCS-1 ASN.1 module. In this case, the values do not depend on any generated or built-in class definition, they are simply lists of integers.

Class Definitions

Python class definitions are generated for the following ASN.1 types:

- Constructed types (SEQUENCE/SET, SEQUENCE OF/SET OF, CHOICE)
- Primitive types with constraints
- Enumerated types
- BIT STRING types having named bits

The class definition for a composite type such as a SEQUENCE contains a constructor, encode/decode methods for different encoding rules, and other methods depending on selected compilation options. For example, if -print is specified, an __str__ method is generated to create a string representation of data in the class, and a print_value method is added to print the value to standard output.

In the other cases, the class contains only static methods that call the underlying primitive encode or decode function and then check the constraint values. For ENUMERATED, a dict is generated that associates identifier names with integer values. For a BIT STRING with named bits, a class is generated to enable bit manipulation by bit name.

Complex Value Definitions

Complex value definitions are values formed from composite types such as SEQUENCE or SET that depend on class definitions. The value definitions are created by creating an instance of the type class and then sequentially populating each element with the assigned value.

Chapter 5. ASN.1 Type to Python Class Mappings

The following sections discuss the specific mappings of ASN.1 types to Python classes. In the case of most primitive types, classes are not generated. Values in the supported value formats are encoded directly.

For all types a class is generated if the type is tagged; e.g.:

```
TaggedBMP ::= [APPLICATION 2] BMPString
```

This statement is true even if the description below for a particular type says no class is generated for the type or doesn't mention the type being tagged as a condition that will cause a class to be generated.

BIT STRING

There are several ways an ASN.1 BIT STRING may be represented in Python:

- `osyspyrt.asn1univtype.Asn1BitString`

  This is the type that decoding will produce by default. It combines binary data with the number of bits in the BIT STRING. This class provides attributes and methods useful in manipulating the value either as a sequence of bytes or as a sequence of bits.

- bytes or bytearray object

  Either of these can be used to provide data for encoding. The BIT STRING will include all bits of the given value.

- An instance of a generated class. This applies when the BIT STRING is defined with named bits, as described below.

BIT STRINGs with named bits

A Python class is generated for an ASN.1 BIT STRING defined with named bits. The generated class defines attributes and properties that are useful for working with the value using the named bits. As an example, consider this ASN.1:

```
Operations ::= BIT STRING  { compute-checksum(0), compute-signature(1),
   verify-checksum(2), verify-signature(3), encipher(4), decipher(5), hash(6),
   generate-key(7) }
```

The resulting (elided) class is:

```
class Operations(univ.NamedBitsBase):
    ...
    named_bits = {0: "compute-checksum",
        1: "compute-signature",
        ...
        7: "generate-key"}
        """Named bit map for use with NamedBitsBase."""

    @property
```

```
    def compute_checksum(self):
    """Return true if compute-checksum bit is set."""
    ...

    @compute_checksum.setter
    def compute_checksum(self, value=True):
    """Set compute-checksum bit on/off."""
    ...
```

As you can see, properties are defined with names correponding to each of the named bits. You can use these to set/get the value of each named bit.

Also, the generated class derives from `osyspyrt.asn1univtype.NamedBitBase`, so that it ultimately derives from, and enhances, the behavior of `osyspyrt.asn1univtype.Asn1BitString`. In particular:

- Instances of the generated class can be initialized using list of named bits:

  ```
  xyz = Operations(("encipher", "decipher"))
  ```

- The value of instances of the generated class can be set using a list of named bits:

  ```
  xyz.set(("encipher", "decipher"))
  ```

- The str output will be a list of named bits:

  ```
  xyz = Operations(("encipher", "decipher"))
  print(str(xyz)) # prints "[encipher, decipher]"
  ```

- The get_bit and set_bit methods can be passed a bit name instead of a bit index.

You are not required to use the generated class, but the generated decoders will use it for decoded values.

BOOLEAN

A Python class is generated for an ASN.1 BOOLEAN type only if the type is tagged. This is only true for BER/DER encoding rules; in other cases, no class is generated.

If a class is generated, it will contain static methods for the BER encoding or decoding of a boolean value.

Example ASN.1:

```
B ::= [APPLICATION 2] BOOLEAN
```

Generated Python Class:

```
class B:
    @staticmethod
    def ber_decode(decbuf, explicit=True, impllen=None):
    ...

    @staticmethod
    def ber_encode(value, encbuf, explicit=True):
    ...
```

The value that is returned by the decode method or passed into the encode method is a Python boolean value (True or False).

See the BER encode/decode methods for information on how to call these methods.

INTEGER

An ASN.1 INTEGER assignment results in the creation of a Python class in the following cases:

- The type has a constraint (for example, INTEGER (1..10))
- The type has named numbers

In these cases, a class is generated that contains static encode and decode methods. For encode, the method first checks the value to make sure it is within the bounds of the constraint and then calls the encode_integer function within the base encoding rules library. For decode, the decode_integer function is first called to decode the value and then the value checked against the constraints to determine if it is within the defined range.

In the case of named numbers, tables are generated to map the actual integer values to their numeric equivalents. These allow an integer value to be represented as a string or integer value. If string, the string value is looked up in the table prior to encode. If integer, it is encoded directly. On decode, the decode integer value is looked up in the table to see if it has a named identifier. If it does, the identifier string value is returned; otherwise the integer value.

Example ASN.1:

```
I ::= INTEGER (1..10)
```

Generated Python Class:

```
class I:
    @staticmethod
    def ber_decode(decbuf, explicit=True, impllen=None):
    ...

    @staticmethod
    def ber_encode(value, encbuf, explicit=True):
    ...
```

The value that is returned by the decode method or passed into the encode method may be an integer numeric value, or, in the case of named numbers, may be the string representation of the number.

See the BER encode/decode methods for information on how to call these methods.

ENUMERATED

An ASN.1 ENUMERATED assignment always results in the creation of a Python class. It is very similar to the named number case described for INTEGER above except in this case, if the identifier is not within the defined set, an exception is raised. However, an exception to this rule is if the ENUMERATED type is extended (i.e. has a ... within it). In this case, it is treated the same as a named number since it may have a value outside of the defined set.

OCTET STRING

A Python class is generated for an ASN.1 OCTET STRING type only if the following are true:

- The type is tagged.
- The type has a size constraint (for example, OCTET STRING (SIZE(1..5))

If a class is generated, it will contain static methods for the BER encoding or decoding of the value.

Example ASN.1:

```
Os2 ::= OCTET STRING (SIZE(1..5))
```

Generated Python Class:

```
class Os2:
    @staticmethod
    def ber_decode(decbuf, explicit=True, impllen=None):
    ...

    @staticmethod
    def ber_encode(value, encbuf, explicit=True):
    ...
```

The value that is returned by the decode method or passed into the encode method is a Python bytearray. Also, for encode, the type of the value passed in may be a string. In this case, the ordinal bytes of the characters are encoded.

See the BER encode/decode methods for information on how to call these methods.

Character String Types

ASN.1 character string types include fixed 8-bit character string types (IA5String, PrintableString, NumericString, etc.), variable-length character size types (UTF8String), and 16-bit wide character string types (BMPString) and 32-bit wide character string types (UniversalString). A Python class is generated for these types only if the following are true:

- The type is tagged.
- The type has a size constraint (for example, IA5String (SIZE(1..5))

If a class is generated, it will contain static methods for the BER encoding or decoding of the value.

Example ASN.1:

```
CharStr ::= IA5String (SIZE(1..5))
```

Generated Python Class:

```
class CharStr:
    @staticmethod
    def ber_decode(decbuf, explicit=True, impllen=None):
    ...

    @staticmethod
    def ber_encode(value, encbuf, explicit=True):
    ...
```

The value that is returned by the decode method or passed into the encode method is a Python string.

See the BER encode/decode methods for information on how to call these methods.

Time String Types

A Python class is generated for an ASN.1 time string type (GeneralizedTime or UTCTime) only if the type is tagged. This is only true for BER/DER encoding rules; in other cases, no class is generated.

If a class is generated, it will contain static methods for the BER encoding or decoding of a time string value.

Example ASN.1:

```
GT ::= [APPLICATION 2] GeneralizedString
```

Generated Python Class:

```
class GT:
    @staticmethod
    def ber_decode(decbuf, explicit=True, impllen=None):
    ...

    @staticmethod
    def ber_encode(value, encbuf, explicit=True):
    ...
```

The value that is returned by the decode method or passed into the encode method is a Python string value.

Note that the newer ASN.1 time types such as TIME, DATE, TIME-OF-DAY, etc. as documented in ITU-T X.680 Clause 38 are not supported in this release.

See the BER encode/decode methods for information on how to call these methods.

REAL

A Python class is not generated for the ASN.1 REAL type. REAL values are represented using:

- float

  Used to represent any base-2 REAL value, +/-INF, or NaN.

- decimal.Decimal

  Used to represent any base-10 REAL value, +/-INF, or NaN.

Note that the ASN.1 data model divides all mathematical real values, other than zero, into two sets of values, base-10 and base-2 values. The unconstrained REAL type allows values from both sets, but REAL can be constrained to allow values from one set only. Encoding rules typically encode base-10 and base-2 values differently. Decoders will use the Python type corresponding to how the value was encoded (base-2 or base-10). For encoding, if the REAL type was constrained to only base-2 or only base-10 values, use float or decimal.Decimal, respectively.

OBJECT IDENTIFIER and RELATIVE-OID

A Python class is generated for an ASN.1 OBJECT IDENTIFIER or RELATIVE-OID type only if the type is tagged. This is only true for BER/DER encoding rules; in other cases, no class is generated.

If a class is generated, it will contain static methods for the BER encoding or decoding of an OID value.

Example ASN.1:

```
OID ::= [APPLICATION 2] OBJECT IDENTIFIER
```

Generated Python Class:

```
class OID:
    @staticmethod
    def ber_decode(decbuf, explicit=True, impllen=None):
    ...

    @staticmethod
    def ber_encode(value, encbuf, explicit=True):
    ...
```

The value that is returned by the decode method or passed into the encode method is a Python integer list value. The list contains the arcs for the OID value.

See the BER encode/decode methods for information on how to call these methods.

Chapter 6. Generated BER/DER Encode Methods

For Python, the generation of memory-based, definite length BER (Basic Encoding Rules) encode methods is the only encoding method supported at ths time. This also covers the case of DER (Distinguished Encoding Rules) which also uses definite lengths. The extra canonical checks such as sorting elements in a SET are also done if DER is selected as the encoding rules type.

For each ASN.1 production defined in an ASN.1 source file, a Python encode method *may* be generated. This method will convert a populated variable of the given type into an encoded ASN.1 message.

For primitive types, an encode method is only generated if it is required to alter the encoding of the BER run-time function for that type. The Python BER run-time library contains a set of common run-time base methods in the *Asn1BerEncodeBuffer* class. These functions include support for encoding content as well as adding the universal tags associated with the types as defined in the X.680 standard.

So for simple assignments, the generation of an encode method is not necessary. For example, the following production will not result in the generation of an encode method or even a Python class:

```
X ::= INTEGER
```

In this case, the user must use the standard run-time function in the BER library to encode a a value of the type. In this case, this would be the *encode_integer* function.

However, if the type is altered to contain a tag or constraint, then a Python class with a static custom encode method would be generated:

```
X ::= [APPLICATION 1] INTEGER
```

In this case, special logic is necessary to apply the tag value.

Some types will always cause Python classes with encode methods to be generated. At the primitive level, this is true for the ENUMERATED type. This type will always contain a custom set of enumerated values. All constructed types (SEQUENCE, SET, SEQUENCE/SET OF, and CHOICE) will cause Python classes to be generated that include encode methods.

Run-time and Generated Python Encode Methods

Three types of Python encode functions/methods are available for encoding different types of data. These are:

- Standard base run-time functions. These exist in the *asn1ber.py* run-time module and exist within the *Asn1BerEncodeBuffer* run-time class. Examples are *encode_boolean*, *encode_integer*, *encode_bitstr*, etc.

- Static methods in Python classes for primitive types. Classes are generated for primitive types that have been altered by adding tagging information or constraints. The method name in this case is *ber_encode*.

- Instance methods in Python classes for constructed types. Classes are generated for SEQUENCE, SET, SEQUENCE/SET OF, and CHOICE constructs. The encode method in theis case encodes the instance attributes which correspond to the ASN.1 elements defind in the ASN.1 types. The method name in this case is also *ber_encode* as it was in the static method case.

The signature for a Standard base run-time method in the *Asn1BerEncodeBuffer* is as follows:

```
def encode_* (self, value, explicit=True)
```

where * would be a primitive type name (boolean, integer, etc.).

The *self* argument is a reference to the *Asn1BerEncodeBuffer* object from which it was invoked (note that in Python, this argument is not explicitly passed, it refers to the object invoking the method). The *value* argument is the value to be encoded. The *explicit* argumnet is a boolean argument indicating if the universal tag and length associated with the primitive type should be applied on top of the encoded content.

The return value is the length in octets of the encoded message component. Unlike the C/C++ version, a negative value is never returned to indicate an encoding failure. That is handled by the Python exception mechanism.

The signature for a static BER encode method in a class generated for a primitive type is as follows:

```
@staticmethod
def ber_encode(value, encbuf, explicit=True):
```

In this case, there is no 'self' argument. A reference to the encode buffer object is passed as a formal argument in the 2nd position. The value and explicit arguments are as they were in standard base run-time method case and the return value is also the same (the encoded length).

The signature for an instance BER encode method in a class generated for a constructed type is as follows:

```
def ber_encode(self, encbuf, explicit=True)
```

The *self* argument is a reference to an object of the generated class (note that in Python, this argument is not explicitly passed, it refers to the object invoking the method). >The *encbuf* argument is a reference to BER encode buffer object to be used to encode the data. The explicit argument is that same as in the other two cases.

Populating Generated Variables for Encoding

Populating attributes of instances of classes generated for constructed types can be done by creating an instance of the class and then directly populating each attribute. For primitive types, nothing needs to be populated. The values to be encoded are passed directly to the methods.

For classes generated for SEQUENCE/SET OF constructs, the standard Python list structure is used. The typical method is to create an empty list (x = []) and then append items to it.

The only primitive type that is an instance of a class having multiple attributes is BIT STRING. In this case, the *Asn1BitString* class contains an attribute named *numbits* to specify the number of bits in the string and a *value* to hold the actual data.

Some primitive type encode methods can accept data in multiple forms. An example of this is the encode method generated for an INTEGER with named numbers. In this case, the encode method will accept an argument of type string which is assumed to be a named number identifier or it can accept an argument of type int which is the integer value to be encoded.

Procedure for Calling Python BER Encode Methods

The procedure to call a Python encode method for the different cases described above is basically the same. It involves the following three steps:

1. Create an encode message buffer object into which the value will be encoded.

2. Invoke the encode method.

3. Invoke encode message buffer methods to access the encoded message component.

The first step is the creation of an encode message buffer object. This is done by simply creating an instance of the Asn1BerEncodeBuffer class:

```
    encbuf = Asn1BerEncodeBuffer()
```

The encode method is then invoked. In the simple case of a primitive type with no generated class, this involves invoking one of the methods defined in the encode buffer class to encode a primitive value. For example, to encode an integer, one would do:

```
enclen = encbuf.encode_integer(10)
```

This would encode the integer value 10 and add the universal tag and length bytes (the explict argument is set to True by default).

The procedure for invoking a static method is similar. The general form is <classname>.encode(value, encbuf, explicit). So, for example, a class named EmployeeNumber would be generated for the following definition:

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

To encode an employee number of 51, one would do the following:

```
EmployeeNumber.encode(51, encbuf)
```

This would encode value 51 and add the APPLICATION 2 tag.

Finally, to invoke the instance method in the class generated for a constructed type, one would first populate the attribute values and then invoke the encode method. To encode an instance of the Name class in the employee sample, one would first create an instance of the class, populate the attributes, and then invoke the encode method:

```
jSmithName = Name()
jSmithName.givenName = 'John'
jSmithName.initial = 'P'
jSmithName.familyName = 'Smith'

jSmithName.encode(encbuf)
```

This will encode the full name and add the assigned tag.

The final step once the data is encoded is to retrieve a reference to it from the encode buffer object. This is done using the *buffer* method. The encoded data is returned in the form of an in-memory byte array.

A complete example showing how to invoke an encode method is as follows:

```
   # Note: personnelRecord object was previously populated with data

   # Step 1: Create an encode buffer object

   encbuf = Asn1BerEncodeBuffer()

   # Step 2: Invoke the encode method. Note that it must be done
   # from within a try/catch block..
```

```
try:
    personnelRecord.encode(encbuf, True);

    if (trace):
        # dump encoded message
        print(encbuf.bin_dump())

    # Step 3: Access the encoded message component. In this
    # case, we use methods in the class to write the component
    # to a file and output a formatted dump to the message.dmp
    # file..

    # Write the encoded record to a file
    f = open('message.dat', 'wb')
    f.write(encbuf.buffer())
    f.close()

    # Generate a hex dump file for comparisons
    f = open('message.hex', 'w')
    f.write(hexdump(encbuf.buffer()))
    f.close()

except Exception:
    tb = traceback.format_exc()
    print(tb)
```

Chapter 7. Generated BER/DER Decode Methods

For Python, BER decode methods are generated that support decoding BER data in either definite or indefinite length from. Data may be read from any valid streaming data source supported by Python including memory, files, and sockets. Decoding data in DER and CER format is also supported as these are subsets of BER, however, checks are not done to ensure this data in in proper canonical format as specified by these rules.

For each ASN.1 production defined in an ASN.1 source file, a Python decode method *may* be generated. This method will read data from the stream specified when the decode buffer object was created and decode into variables in memory.

For primitive types, a decode method is only generated if it is required to alter the BER encoding the standard run-time function for that type would produce. The Python BER run-time library contains a set of common run-time base methods in the *Asn1BerDecodeBuffer* class. These functions include support for decoding content as well as the universal tags associated with the types as defined in the X.680 standard.

So for simple assignments, the generation of a decode method is not necessary. For example, the following production will not result in the generation of an decode method or even a Python class:

```
X ::= INTEGER
```

In this case, the user must use the standard run-time function in the Asn1BerDecodeBuffer class in the BER run-time library to decode a a value of the type. In this case, this would be the *decode_integer* function.

However, if the type is altered to contain a tag or constraint, then a Python class with a static custom decode method would be generated:

```
X ::= [APPLICATION 1] INTEGER
```

In this case, special logic is necessary to parse the tag value.

Some types will always cause Python classes with decode methods to be generated. At the primitive level, this is true for the ENUMERATED type. This type will always contain a custom set of enumerated values. All constructed types (SEQUENCE, SET, SEQUENCE/SET OF, and CHOICE) will cause Python classes to be generated that include decode methods.

Run-time and Generated Python Decode Methods

Three types of Python decode functions/methods are available for decoding different types of data. These are:

- Standard base run-time functions. These exist in the *asn1ber.py* run-time module and exist within the *Asn1BerDecodeBuffer* run-time class. Examples are *decode_boolean*, *decode_integer*, *decode_bitstr*, etc.
- Static methods in Python classes for primitive types. Classes are generated for primitive types that have been altered by adding tagging information or constraints. The method name in this case is *ber_decode*.
- Instance methods in Python classes for constructed types. Classes are generated for SEQUENCE, SET, SEQUENCE/SET OF, and CHOICE constructs. The decode method in theis case decodes data into the instance attributes which correspond to the ASN.1 elements defind in the ASN.1 types. The method name in this case is also *ber_decode* as it was in the static method case.

The signature for a Standard base run-time method in the *Asn1BerDecodeBuffer* is as follows:

```
def decode_* (self, explicit=True, impllen=None)
```

where * would be a primitive type name (boolean, integer, etc.).

The *self* argument is a reference to the *Asn1BerDecodeBuffer* object from which it was invoked (note that in Python, this argument is not explicitly passed, it refers to the object invoking the method). The *explicit* argument is a boolean argument indicating if the universal tag and length associated with the primitive type should be parsed prior to decoding the content.

The *impllen* argument is a length oject that only has meaning if explicit is False. In this case, it specified the length of the contents field to be decoded. If explicit is True, this length is parsed from the encoded tag/length value that precedes the content.

The return value is the decoded content value. This will be of whatever type the content being decoded is as specified in the ASN.1 schema. Decoding errors are handled by the Python exception mechanism.

The signature for a static BER decode method in a class generated for a primitive type is as follows:

```
    @staticmethod
    def ber_decode(decbuf, explicit=True, impllen=None):
```

In this case, there is no 'self' argument. A reference to the decode buffer object is passed as a formal argument in the 2nd position. The explicit and impllen arguments are as they were in standard base run-time method case and the return value is also the same (the decoded value).

The signature for an instance BER decode method in a class generated for a constructed type is as follows:

```
    def ber_decode(self, decbuf, explicit=True, impllen=None)
```

The *self* argument is a reference to an object of the generated class (note that in Python, this argument is not explicitly passed, it refers to the object invoking the method). >The *decbuf* argument is a reference to BER decode buffer object to be used to encode the data. The explicit and impllen arguments are that same as in the other two cases.

Procedure for Calling Python BER Decode Methods

The procedure to call a Python decode method for the different cases described above is basically the same. It involves the following steps:

1. Create a decode message buffer object that describes the stream from which the value will be decoded.

2. Invoke the decode method.

The first step is the creation of a decode message buffer object. This is normally done by invoking a class method to to specify the data source. Two standard methods that can be used for this purpose are *from_bytes* to specify a memory source or *from_file* to specify a file source. For example:

```
    decbuf = Asn1BerDecodeBuffer.from_file('message.dat')
```

The decode method itself is then invoked. In the simple case of a primitive type with no generated class, this involves invoking one of the methods defined in the decode buffer class to decode a primitive value. For example, to decode an integer, one would do:

```
value = decbuf.decode_integer()
```

. This would decode the content at the current decode buffer cursor position which is expected to be a tag-length-value (TLV) with integer universal tag and integer content.

The procedure for invoking a static method is similar. The general form is <classname>.decode(decbuf, explicit, implen). So, for example, a class named EmployeeNumber would be generated for the following definition:

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

To decode an employee number, one would do the following:

```
value = EmployeeNumber.decode(decbuf)
```

This would check to ensure the tag value at the current decode buffer position is APPLICATION 2 tag and then parse the length and decode the content. If successful, the decoded integer value will be assigned to value data variable.

Finally, to invoke the instance method in the class generated for a constructed type, one would first create an instance of the class and then invoke the generated decode method. To decode an instance of the Name class in the employee sample, one would first create an instance of the class and then invoke the decode method:

```
jSmithName = Name()
jSmithName.decode(decbuf)
```

If successful, the attributes of the jSmithName instance will be populated with the decoded values.

A complete example showing how to invoke a decode method is as follows:

```
   try:

      # Step 1: create a decode message buffer object to describe the
      # message to be decoded. This example will use a file input
      # stream to decode a message directly from a binary file..

      decbuf = Asn1BerDecodeBuffer.from_file(filename)

      # Step 2: create an object of the generated type and invoke the
      # decode method..

      personnelRecord = PersonnelRecord()
      personnelRecord.decode(decbuf)

      # Step 3: process the data

      if trace:
         print("Decode was successful")
         personnelRecord.print_value("personnelRecord", 0);


   except Exception:
      tb = traceback.format_exc()
      print(tb)
```

Chapter 8. Generated JER (JSON) Decode Methods

Recall from the "ASN.1 Type to Python Class Mappings" chapter that, for many ASN.1 types, a built-in Python type or a type from the ASN1C Python run-time is used to represent values of the ASN.1 type. A Python class is generated for an ASN.1 type when either a) there is no such Python type that can be used to represent the value, OR b) the ASN.1 type is assigned a name and its definition includes constraints, tags, or encoding instructions that alter the type's value space or encoding from the basic ASN.1 type, such as for `MyInteger ::= INTEGER(0..8)`

For each ASN.1 type defined in an ASN.1 source file and for which a Python class is generated, a Python decode method *may* be generated (when an appropriate method is inherited, it is not necessary to generate a decode method).

When a Python class *is* generated, it is generated to either represent the ASN.1 type (such as for `SEQUENCE`) or to simply provide encode and decode methods. In the former case, the generated methods will be instance methods, while in the latter case the generated methods will be static methods. Naturally, decode methods which are instance methods will decode into the object on which the decode method is invoked. Decode methods which are static methods will return the result of decoding.

The Python JER run-time library provides decoding of basic ASN.1 types through the `Asn1JsonDecodeBuffer` class. The `Asn1JsonDecodeBuffer` class decodes data from any io.TextIOBase object.

If a Python class is NOT generated for a given ASN.1 type, the methods in `Asn1JsonDecodeBuffer` can be used to decode it. For example, no class will be generated for:

```
MyInteger ::= INTEGER
```

while a class with static methods will be generated for:

```
MyInteger ::= INTEGER(0..8)
```

Run-time and Generated Python Decode Methods

Three types of Python decode functions/methods are available for decoding different types of data. These are:

- Standard base run-time functions. These exist in the `asn1json.py` run-time module, in the `Asn1JsonDecodeBuffer` run-time class. Examples are `decode_bitstring` and `decode_real`. For some types, which are closely aligned with the JSON representation, read* methods are used, such as `read_boolean` and `read_string`. These methods return a representation of the decoded value.

- Static methods in Python classes, such as for primitive types and `SEQUENCE/SET OF`, as described above. The method name in this case is `json_decode`. The return value will be the Python class that represents values of the ASN.1 type, whether that is a built-in class or from the ASN1C Python runtime.

- Instance methods in Python classes for constructed types. Classes are generated for `SEQUENCE`, `SET`, and `CHOICE` constructs. The decode method in this case decodes into the instance attributes which correspond to the ASN.1 elements defind in the ASN.1 types. The method name is also `json_decode`, as it was in the static method case. There is no return value.

The generated decode methods accept an `Asn1JsonDecodeBuffer`, the buffer to decode from. Example signatures are:

```
@staticmethod
def json_decode(decbuf):


def json_decode(self, decbuf):
```

All decode methods raise an exception if decoding fails.

Procedure for Calling Python JER Decode Methods

The procedure to call a Python decode method for the different cases described above is basically the same. It involves the following steps:

1. Create a decode message buffer object on the data to be decoded, from which the value will be decoded.

2. Invoke the decode method.

The first step is the creation of a decode message buffer object. An `Asn1JsonDecodeBuffer` can be created on any io.TextIOBase object, or you can use the convenience class method `from_text` to create one from a string of JSON text. For example:

```
# Create buffer on a JSON text file
decbuf = Asn1JsonDecodeBuffer(open(filename, 'r'))

# Create a buffer on JSON text
decbuf = Asn1JsonDecodeBuffer.from_text(json_text)
```

The decode method itself is then invoked. In the simple case of a primitive type with no generated class, this involves invoking one of the methods defined in the decode buffer class to decode a primitive value. For example, to decode an integer, do:

```
value = decbuf.read_int()
```

This would decode the content at the current decode buffer cursor position, which is expected to be a JSON number representing an integer.

The procedure for invoking a static method is similar. The general form is <classname>.json_decode(decbuf). For example, a class named `EmployeeNumber` would be generated for the following definition:

```
EmployeeNumber ::= INTEGER(0..100)
```

To decode an employee number, do the following:

```
value = EmployeeNumber.json_decode(decbuf)
```

This would decode the JSON number and check to ensure the constraint is satisfied. If successful, the decoded integer value will assigned to `value`.

Finally, to invoke an instance method in a class generated for a constructed type, first create an instance of the class and then invoke the generated decode method:

```
jSmithName = Name()
jSmithName.json_decode(decbuf)
```

If successful, the attributes of the jSmithName instance will be populated with the decoded values.

A complete example showing how to invoke a decode method is as follows:

```
try:
    # Step 1: create a decode buffer.  This example uses a file for input.
    f = open(in_filename, 'r')
    decbuf = Asn1JsonDecodeBuffer(f)

    # Step 2: create an object of the generated type and invoke the decode
    # method.
    personnelRecord = PersonnelRecord()
    personnelRecord.json_decode(decbuf)

    f.close()

    personnelRecord.print_value("personnelRecord", 0)

except Exception:
    print(traceback.format_exc())
    sys.exit(-1)
```

Chapter 9. Generated JER (JSON) Encode Methods

For each ASN.1 production defined in an ASN.1 source file, a Python encode method *may* be generated. This method will convert a populated variable of the given type into an encoded ASN.1 message.

Recall from the "ASN.1 Type to Python Class Mappings" chapter that, for many ASN.1 types, a built-in Python type or a type from the ASN1C Python run-time is used to represent values of the ASN.1 type. A Python class is generated for an ASN.1 type when either a) there is no such Python type that can be used to represent the value, OR b) the ASN.1 type is assigned a name and its definition includes constraints, tags, or encoding instructions that alter the type's value space or encoding from the basic ASN.1 type, such as for `MyInteger ::= INTEGER(0..8)`

When a Python class *is* generated, it is generated to either represent the ASN.1 type (such as for `SEQUENCE`) or to simply provide encode and decode methods. In the former case, the generated methods will be instance methods, while in the latter case, the generated methods will be static methods.

The Python JER run-time library provides encoding of basic ASN.1 types through the `Asn1JsonEncodeBuffer` class. If a Python class is NOT generated for a given ASN.1 type, the methods in `Asn1JsonEncodeBuffer` can be used to encode it. For example, no class will be generated for:

```
MyInteger ::= INTEGER
```

while a class with static methods will be generated for:

```
MyInteger ::= INTEGER(0..8)
```

Run-time and Generated Python Encode Methods

Three types of Python encode functions/methods are available for encoding different types of data. These are:

- Standard base run-time functions. These exist in the `asn1json.py` run-time module, in the `Asn1JsonEncodeBuffer` run-time class. Examples are `encode_bitstring` and `encode_real`, but for many types, `write` or `write_string` are all that is needed.
- Static methods in Python classes, such as for primitive types and `SEQUENCE/SET OF`, as described above. The method name in this case is `json_encode`.
- Instance methods in Python classes for constructed types. Classes are generated for `SEQUENCE`, `SET`, and `CHOICE` constructs. The encode method in this case encodes the instance attributes which correspond to the ASN.1 elements defind in the ASN.1 types. The method name is also `json_encode`, as it was in the static method case.

The encode methods do not have a return value and will raise an exception if the encoding fails.

The encode methods on `Asn1JsonEncodeBuffer` have a value parameter for the value to be encoded, and in some cases additional parameters (e.g. `encode_octetstring` has a parameter for the base for the encoding, `base`). Of course, these methods are invoked on an instance of `Asn1JsonEncodeBuffer`.

The generated encode methods accept an `Asn1JsonEncodeBuffer`, the buffer to encode to. Static methods additionally accept the value to be encoded. Some examples:

```
  @staticmethod
  def json_encode(encbuf, value):


  def json_encode(self, encbuf):
```

Procedure for Calling Python JER Encode

The procedure to call a Python encode method for the different cases described above is basically the same. It involves the following three steps:

1. Create an encode message buffer object into which the value will be encoded.

2. Invoke the encode method.

The first step is the creation of an encode message buffer object. This is done by simply creating an instance of the Asn1JsonEncodeBuffer class:

```
  # Encode to a file
  encbuf = Asn1JsonEncodeBuffer(filename="filename")

  # Or, encode to a TextIOBase object, such as StringIO
  encbuf = Asn1JsonEncodeBuffer(writer=text_io_object)
```

The encode method is then invoked. In the simple case of a primitive type with no generated class, this involves invoking one of the methods defined in the encode buffer class to encode a primitive value. For example, to encode an integer, one would do:

```
encbuf.write(str(value))
```

The procedure for invoking a static method is similar. The general form is <classname>.json_encode(encbuf, value). So, for example, a class named EmployeeNumber would be generated for the following definition:

```
EmployeeNumber ::= INTEGER(0..100)
```

To encode an employee number of 51, one would do the following:

```
EmployeeNumber.json_encode(encbuf, 51)
```

This would verify the constraint was satisfied and encode the value 51.

Finally, to invoke the instance method in the class generated for a constructed type, one would first populate the attribute values and then invoke the encode method. To encode an instance of the Name class in the employee sample, one would first create an instance of the class, populate the attributes, and then invoke the encode method:

```
jSmithName = Name()
jSmithName.givenName = 'John'
jSmithName.initial = 'P'
jSmithName.familyName = 'Smith'

jSmithName.json_encode(encbuf)
```

This will encode the name.

A complete example showing how to invoke an encode method is as follows:

```
# Note: jSmithPR object was previously populated with data

try:
    jSmithPR = PersonnelRecord()

    # Step 1: Create an encode buffer object
    encbuf = Asn1JsonEncodeBuffer(filename=out_filename)

    # Step 2: Invoke the encode method.
    jSmithPR.json_encode(encbuf)

    encbuf.close()

except Exception:
    print(traceback.format_exc())
    sys.exit(-1)
```

Chapter 10. Generated Sample Programs

The –*writer* and -*reader* options cause writer and reader sample programs to be generated.

The writer program contains sample code to populate and encode an instance of ASN.1 data. The main purpose is to provide a code template to users for writing code to populate objects. This is quite useful to users because generated objects can become very complex as the ASN.1 schemas become more complex. The writer code also shows users how to instantiate an encode buffer object and how to use encode functions. The writer program writes the encoded data to a file. If the writer program is generated by using both -*writer* and -*gentest* options, then the generated writer program uses random data to populate the object(s).

The reader program on the other hand reads the encoded data from a file. It shows users how to use a decode buffer object to decode data and populate the corresponding class object. On successful decode, it prints the decoded data to standard output.

Chapter 11. Generated Print Methods

The *-print* or *-genprint* option causes printing methods to be generated. These functions can be used to print the contents of variables of generated types. The printing methods are generated in each of the generated Python classes that encapsulate multiple members.

Generated Python print_value Method Format and Calling Parameters

The print_value method is provided in with the following signature:

```
def print_value(self, elem_name="<name>", indent=0):
```

The *elem_name* argument is used to specify the top-level variable name of the item being printed. In the generated code this is defaulted to the name of the item in the ASN.1, but in your call to print_value() you can specify a different value.

The *indent* argument is used to specify the indentation level for printing nested types. The generated code sets 0 as the default value for this argument, but you can specify a different value. Each indentation level results in an indentation of 3 spaces in the printed output. So for example specifying 1 for this argument means the outermost lines will be indented by 3 spaces, the next level 6 spaces, and so on. Allowing the value to default to 0 means the outermost lines will not be indented, the next level will be indented by 3 spaces, and so on.

For example, the simplest call to print the personnelRecord from the previous examples would be as follows:

```
personnelRecord.print_value ();
```

The output would be formatted as follows:

```
PersonnelRecord {
   name {
      givenName = 'John'
      initial = 'P'
      familyName = 'Smith'
   }
   number = 51
   title = 'Director'
   dateOfHire = '19710917'
   nameOfSpouse {
      givenName = 'Mary'
      initial = 'T'
      familyName = 'Smith'
   }
   children[0] {
      name {
         givenName = 'Ralph'
         initial = 'T'
         familyName = 'Smith'
      }
      dateOfBirth = '19571111'
   }
   children[1] {
```

```
            name {
               givenName = 'Susan'
               initial = 'B'
               familyName = 'Jones'
            }
            dateOfBirth = '19590717'
         }
      }
```

Generated Python __str__ Method Format and Calling Parameters

In addition to the print_value method described above, generated Python code also includes the usual Python __str__ method. The presence of this method allows you to manipulate a generated object as a string. Using the personnelRecord example from above:

```
print(personnelRecord)
```

or:

```
prstring = str(personnelRecord)
print(prstring)
```

The output of either of these print commands would be as follows:

```
{
   name {
      givenName = 'John'
      initial = 'P'
      familyName = 'Smith'
   }
   number = 51
   title = 'Director'
   dateOfHire = '19710917'
   nameOfSpouse {
      givenName = 'Mary'
      initial = 'T'
      familyName = 'Smith'
   }
   children[0] {
      name {
         givenName = 'Ralph'
         initial = 'T'
         familyName = 'Smith'
      }
      dateOfBirth = '19571111'
   }
   children[1] {
      name {
```

```
            givenName = 'Susan'
            initial = 'B'
            familyName = 'Jones'
         }
         dateOfBirth = '19590717'
      }
   }
```

You can see that the output is virtually the same as the output from the print_value call, except there is no name assigned to the entire grouping.

The __str__ method is called by the generated print_value method, but it can also be called directly by you if desired. Its signature is as follows:

```
def __str__(self, elem_name=None, indent_level=None):
```

The elem_name argument is a name to assign to the listing, and the indent_level argument controls the indentation of the outermost level of the listing. Each indent level results in an indentation of 3 spaces. So, again using the personnelRecord example, you can do this:

```
prstring = personnelRecord.__str__("ThisRecord", 1)
print(prstring)
```

The output of the print command would be as follows:

```
ThisRecord {
   name {
      givenName = 'John'
      initial = 'P'
      familyName = 'Smith'
   }
   number = 51
   title = 'Director'
   dateOfHire = '19710917'
   nameOfSpouse {
      givenName = 'Mary'
      initial = 'T'
      familyName = 'Smith'
   }
   children[0] {
      name {
         givenName = 'Ralph'
         initial = 'T'
         familyName = 'Smith'
      }
      dateOfBirth = '19571111'
   }
   children[1] {
      name {
         givenName = 'Susan'
         initial = 'B'
```

```
            familyName = 'Jones'
         }
         dateOfBirth = '19590717'
      }
   }
```

So in this case the output is all indented 3 spaces beyond the indentation from the previous examples, and the name assigned to the listing is "ThisRecord".

Chapter 12. Generated Compare Methods

The *-compare* or *-gencompare*command line option causes an __eq__ method to be added to relevant generated classes. The presence of this method allows the Pyton equality operators to be used to compare instances of generated objects.

For example, if your program has two instances of the generated PersonnelRecord class, named pr1 and pr2, the two instances can be compared for equality using this line:

```
if pr1 == pr2:
```

Chapter 13. Generated Copy Methods

When `-copy` or `-gencopy`is specifiedon the command line, ASN1C will generate `copy_value` methods in Python code. These methods perform a deep copy.

The `copy_value` method is provided in the following form:

```
def copy_value(self):
```