

ASN1C

ASN.1 Compiler User's Guide for Python

Version 7.8

Objective Systems, Inc.

December 2023

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

Copyright © 1997-2023 Objective Systems, Inc.

License. The software described in this document is furnished under a license agreement and may be used only in accordance with the terms of this agreement. This document may be distributed in any form, electronic or otherwise, provided that it is distributed in its entirety with the copyright and this notice intact.

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

1. Overview of ASN1C for Python	1
2. ASN1C Command Line Interface (CLI)	2
Running ASN1C	2
ASN1C Python Command Line Options	2
Compiler Configuration File	5
ASN.1 Standard Revisions	8
Compiler Error Reporting	9
3. ASN1C GUI Users Guide	10
Quick Start	10
Activating a License Key	10
Creating a New Project	12
Creating a Project	14
Creating a New Project	14
Editing a Project	15
Opening a Project	15
Saving a Project	15
Editing Schemas	15
Creating a New Schema File	16
Editing a Schema File	16
Deleting a Schema File	17
Compiling	17
Interface	17
Editor	18
Project Window	19
ASN.1 Tree Window	22
Error Log Window	22
Project Settings	23
4. Generated Python Source Code	34
General Form of a Generated Python Source File	34
Import Statements	34
Simple Value Definitions	34
Class Definitions	34
Complex Value Definitions	35
5. ASN.1 Type to Python Class Mappings	36
BIT STRING	36
BIT STRINGs with named bits	36
BOOLEAN	37
INTEGER	38
ENUMERATED	38
OCTET STRING	38
Character String Types	39
Time String Types	40
REAL	40
OBJECT IDENTIFIER and RELATIVE-OID	40
6. Generated BER/DER Encode Methods	42
<i>Run-time and Generated Python Encode Methods</i>	42
<i>Populating Generated Variables for Encoding</i>	43
<i>Procedure for Calling Python BER Encode Methods</i>	43
7. Generated BER/DER Decode Methods	46
<i>Run-time and Generated Python Decode Methods</i>	46
<i>Procedure for Calling Python BER Decode Methods</i>	47

8. Generated JER (JSON) Decode Methods	49
Run-time and Generated Python Decode Methods	49
Procedure for Calling Python JER Decode Methods	50
9. Generated JER (JSON) Encode Methods	52
Run-time and Generated Python Encode Methods	52
Procedure for Calling Python JER Encode	53
10. Generated Sample Programs	55
11. Generated Print Methods	56
Generated Python print_value Method Format and Calling Parameters	56
Generated Python __str__ Method Format and Calling Parameters	57
12. Generated Compare Methods	60
13. Generated Copy Methods	61

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, Python, or Go. 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
- *ASN1C Go Compiler User's Manual* : Go 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 x680-2021 x680-2015 x680-2008 mixed x208	This option selects the version of ASN.1 syntax to be parsed. 'x680' (the default) refers to modern ASN.1 as specified in the ITU-T X.680-X.690 series of standards. Refer to the section called "ASN.1 Standard Revisions" for more information.
-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 (<code>__eq__</code>) in each generated class.
-config	<filename>	This option is used to specify the name of a file containing configuration information for the source file being parsed. A full discussion of the contents of a configuration file is provided in the section called "Compiler Configuration File".
-copy	None	Generate <code>copy_value</code> methods in all non-trivial classes to produce cloned deep-copy of an object instance.

Option	Argument	Description
-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 <code>__str__</code> method to get a string representation of the object as well as a <code>print_value</code> 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. (Note: -jer+ is no longer supported. Similar functionality is now standard JER.)
-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.
-noacomment	None	This option suppresses the generation of the comment that shows the asn1c

Option	Argument	Description
		command that was used to generate the code.
-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.
-noPLMN	None	If not specified, if the ASN.1 contains a production named PLMNidentity (any case), that production will be treated as an OCTET STRING, regardless of how it may be defined in the ASN.1. But if -noPLMN is specified, this special recognition of PLMNidentity will be disabled, and it will be treated according to its definition in the ASN.1 (frequently TBCD-STRING, although in most cases that is incorrect).
-noUniqueNames	None	Turn off the capability to 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>.

Option	Argument	Description
		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)
-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.
-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.
-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 specific modules 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 values="names" types="names"/>	ASN.1 type or values names are specified as an attribute list	<p>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.</p> <p>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.</p>
<exclude values="names" types="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 <i>include</i> 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 mul-

Name	Values	Description
		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.
<isPLMNidentity/>	n/a	This flag identifies the production as a PLMNidentity. Any production so identified will be treated as an OCTET STRING, regardless of how the production might be defined in the ASN.1.
<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 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.

Name	Values	Description
<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.
<notUsedDecode/>	n/a	Similar to the notUsed flag, except that rather than indicating the element is not used at all, it indicates the element is not used for decoding (i.e. it is still used for encoding). This can be useful for reducing the amount of generated code, when used in conjunction with a production level <include/> (which see). It signals that the element's type does not require a decode function for the sake of this element.
<skip/>	n/a	Deprecated. This is now equivalent to <notUsedDecode/>

ASN.1 Standard Revisions

The `-asnstd` option allows you to choose the revision of the ASN.1 standards you want to generate code for. This section explains the differences for these options.

-asnstd x680

- This is the default option.
- This option indicates ASN1C should follow the latest revision of the X.680 and X.690 series that is supported by the tool.

-asnstd x680-2021

- Follows the 2021 revision of the X.680/X.690 series.
- PER encoding of BIT STRINGs with contents constraints: The BIT STRING is padded to a multiple of 8 bits for aligned PER only.
- JER encoding of BIT STRING and OCTET STRING with contents constraints: an encoder's option using a JSON object with a 'containing' key was added in this revision. The encoder will use this encoder's option and the decoder will decode values encoded using either option.

-asnstd x680-2015

- Follows the 2015 revision of the X.680/X.690 series.
- PER encoding of BIT STRINGs with contents constraints: There is no padding of the BIT STRING value.

-asnstd x680-2008

- Follows the 2008 revision of the X.680/X.690 series.
- PER encoding of BIT STRINGs with contents constraints: The BIT STRING is padded to a multiple of 8 bits for both aligned and unaligned PER.

-asnstd mixed

- Used when source files contain modules with both X.208- and X.680-based syntax.

-asnstd x208

- This option supports the deprecated X.208 and X.209 standards.
- Allows use of type 'ANY' (not part of X.680).
- Allows unnamed fields in SEQUENCE, SET, and CHOICE constructs (not allowed by X.680).
- Allows use of ROSE OPERATION and ERROR macros and SNMP OBJECTTYPE macros (macros are not a feature of X.680).

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.

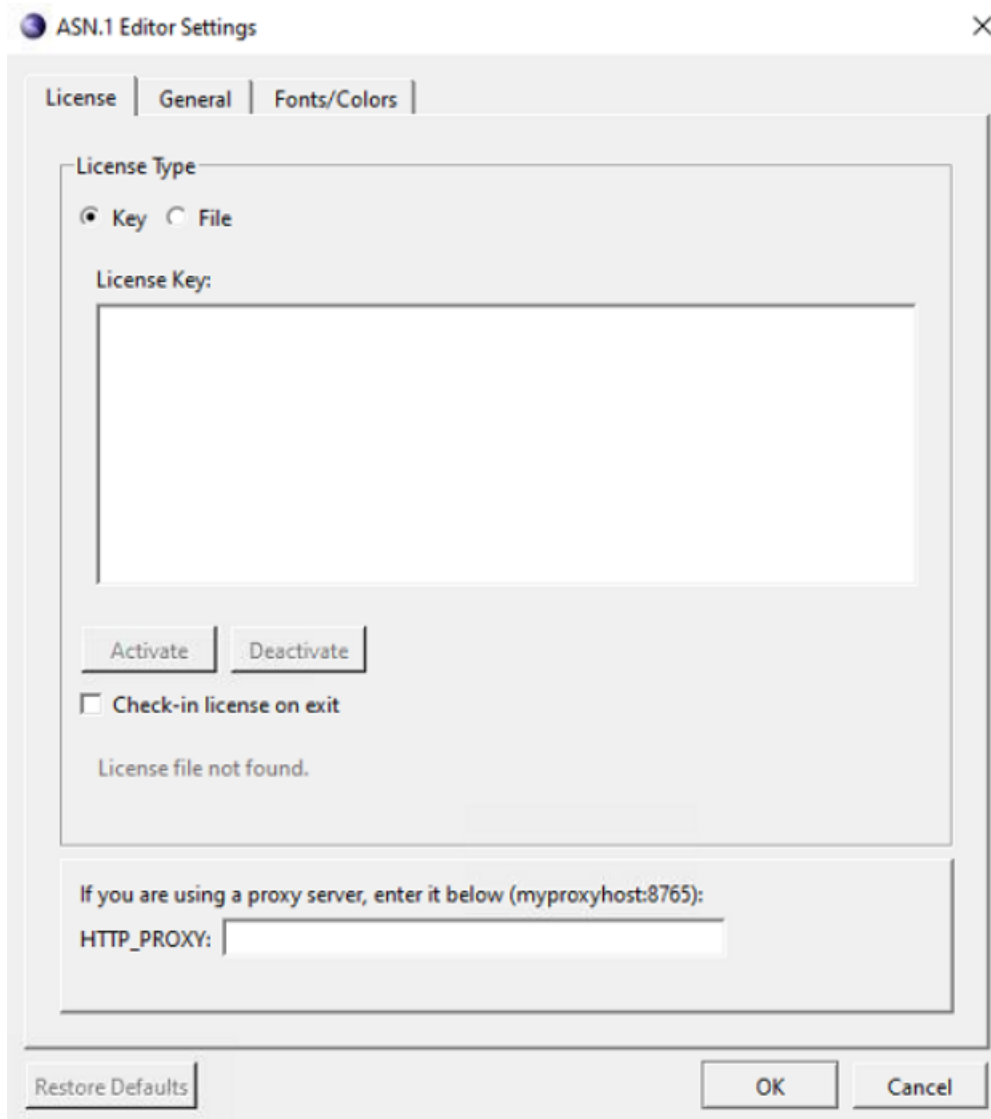
Chapter 3. ASN1C GUI Users Guide

Quick Start

This section demonstrates running ACGUI, activating a license key, creating a new ASN.1 schema, and compiling it to C for BER data. The process is similar for other languages.

Activating a License Key

First, start ACGUI. The Enter License Key window may be displayed to activate a license key.



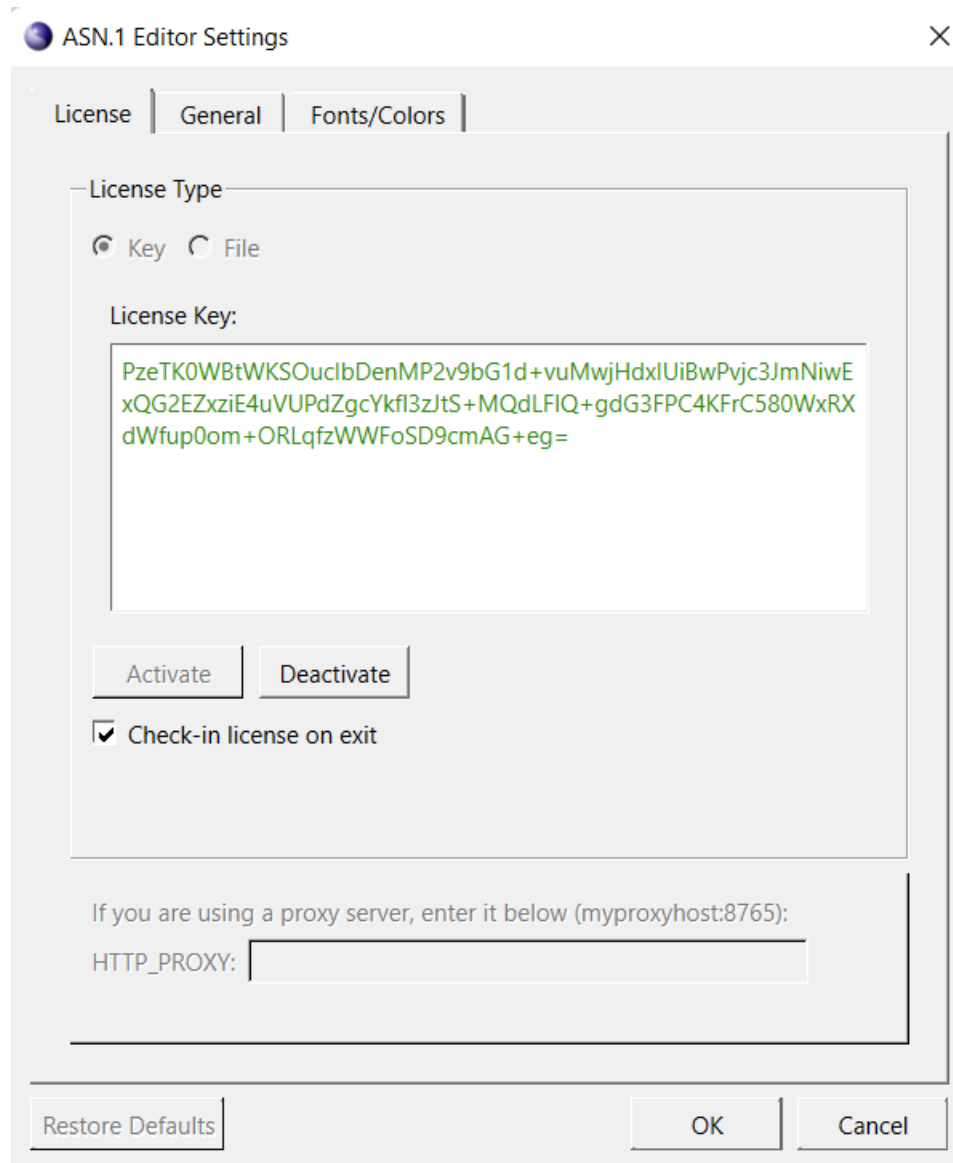
If the Enter License Key window is displayed and it is not showing a current license key, right-click in the text box and paste the accurate license key. Then click Activate to unlock ASN1C.

In some cases, the Enter License Key window is displayed and showing a current license key. In these cases, it is likely that the key being shown is expired. First deactivate the current key by clicking Deactivate. Then, right-click in the text box and paste the current license key, and click Activate to unlock ASN1C.

If an osyslic.txt license file is being used instead of a key, and the osyslic.txt is in a location where the GUI does not look, click Import to find the file and use it to unlock ASN1C.

If Skip is clicked within the Enter License Key window, the features of the GUI can be explored, but code cannot be generated.

If a new license key must be activated and the current one is still valid (for example, if ASN1C is purchased before the evaluation key expires), the existing license must be deactivated first. This deactivation can be done from the GUI by navigating to Tools > Options and then selecting the License tab. The ASN.1 Editor Settings window is displayed.



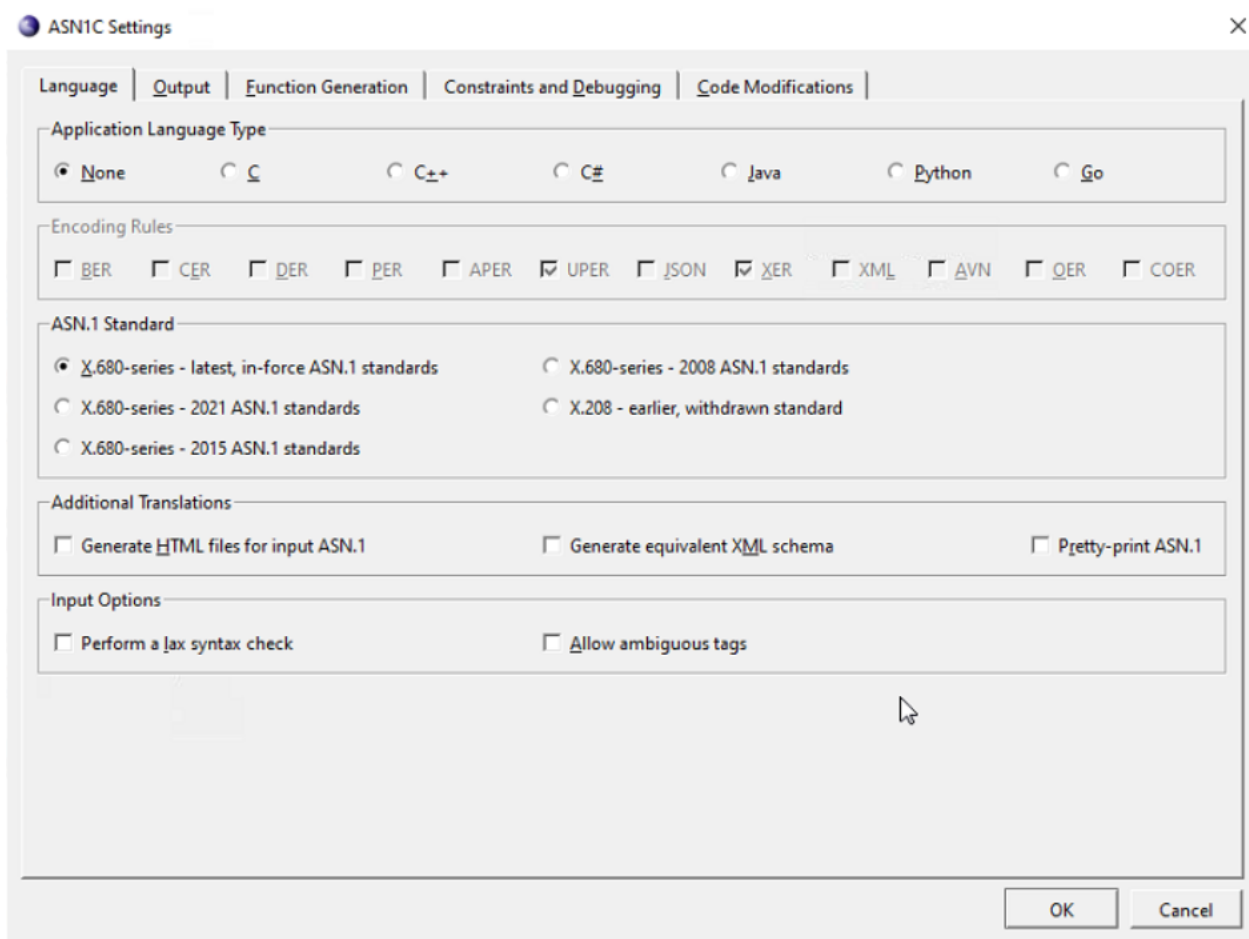
Click Deactivate to deactivate the existing license. Then click OK. Next, navigate to Tools > Options > License tab again to activate the new key.

The "Check-in license on exit" check box is used to indicate that the license should be immediately returned to the license pool upon exit making it available for other users on different machines. If not checked, the machine on which it is being used will continue to hold it until it times out (typically in 24 hours). If you will only be using ASNIC on a single machine most of the time, it is better to keep it unchecked as it will lead to faster startups since the Internet check will not need to be done each time.

The HTTP PROXY box can be used if you are using ASNIC on a machine that requires Internet requests go through a proxy server.

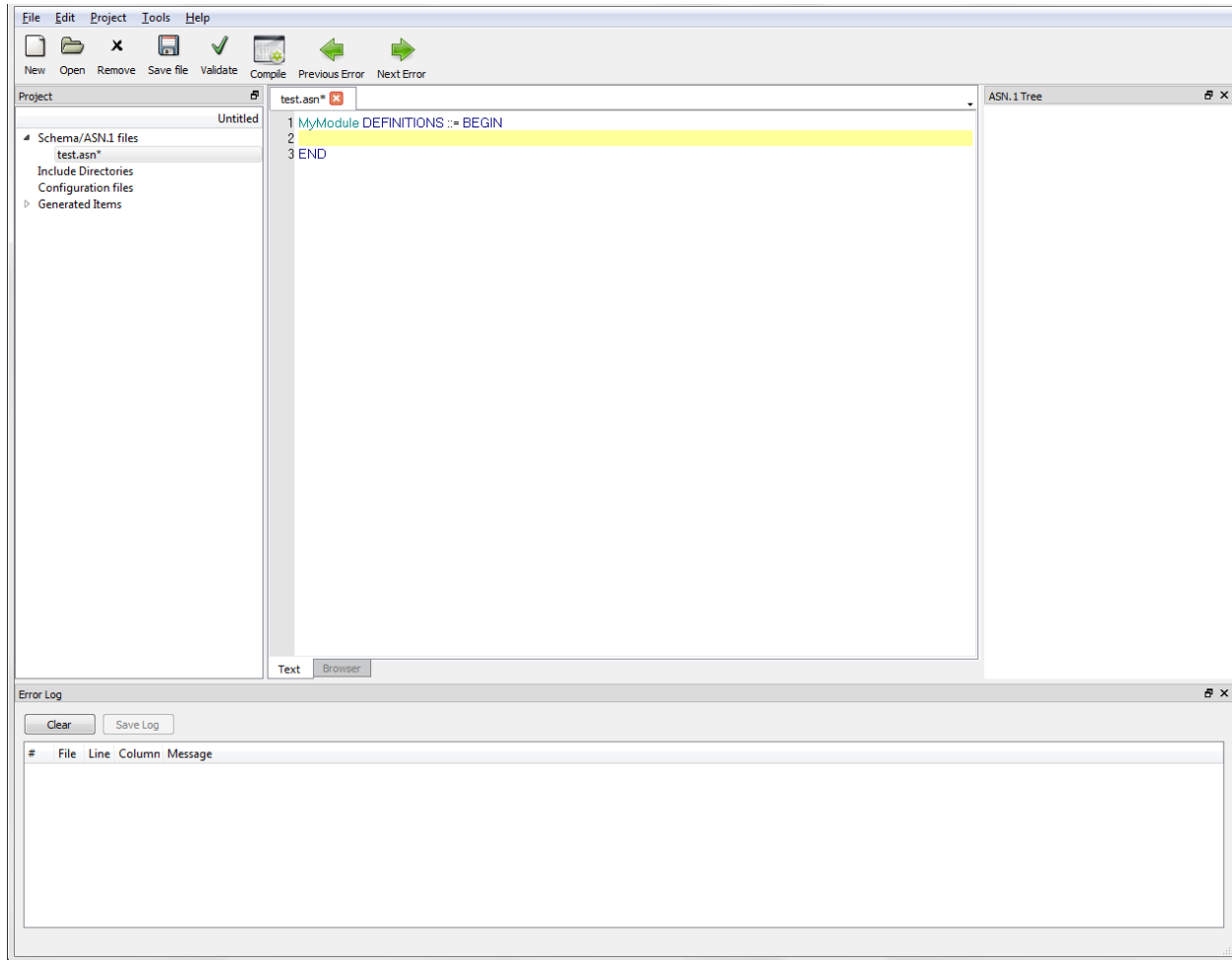
Creating a New Project

Once the ASNIC program has been activated, a new project can be created to store all of the settings. To do this, select Project > New Project from the menu. The ASNIC Settings window is displayed.



The Language tab is displayed by default. In the Application Language Type section, select C. In the Encoding Rules section, select BER. Finally, on the Output tab, in the Output Directory text box, enter or browse to the location where the generated files should be saved. When finished defining the project settings, click OK. These settings can be changed at any time by selecting Project > Project Settings from the menu.

Next, a new schema file is created for the project. Click New in the toolbar or navigate to File > New Schema File. A dialog box is displayed to define a name for the new file. Once entered, the file is added to the project window under the "Schema/ASN.1 files" heading and its empty contents are shown in the editor.



The schema is then written between the "DEFINITIONS ::= BEGIN" and "END" statements in the file. For this example, the following can be entered:

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

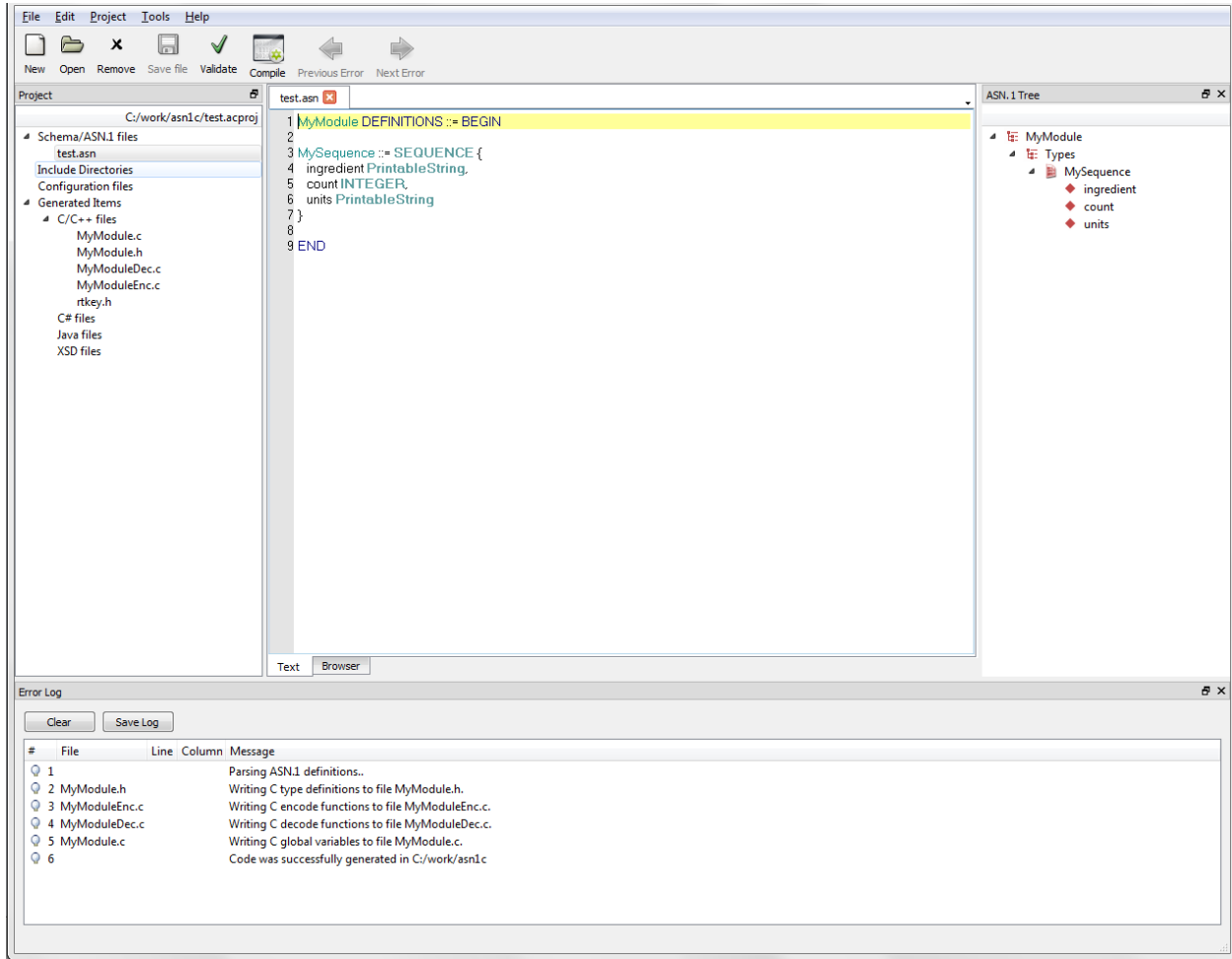
Once the schema is created, click Validate to perform a check for errors.

Note

If the new schema file has not yet been saved, ACGUI asks if it should be. Once saved, ACGUI validates the file.

If the schema has errors, they are displayed in the log at the bottom of the ACGUI window.

Once the project has been configured, click Compile to generate code according to the project settings. If compilation is successful, a list of generated files, sorted according to the selected language, is displayed under the Generated Items heading in the Project pane. If compilation is not successful, any errors are displayed in the Error Log.



At this time, 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 is implicitly created and can be saved to a file at a later time.

Creating a New Project

To create a new project file, select Project > New Project from the menu. The ASN1C Settings window is displayed.

The ASN1C Settings window contains standard tabs for Language, Output, Function Generation, Constraints and Debugging, and Code Modifications. Additional tabs are loaded once an Application Language Type is selected on the Language tab.

Once all project settings have been defined, click OK.

Note

Details regarding the tabs and contents within the ASN1C Settings window can be found in the Interface topic of this guide.

Editing a Project

A project's settings can be changed at any time by navigating to Project > Project Settings.

Opening a Project

To open an existing project, navigate to Project > Open Project. Use the File Explorer window to navigate to the desired project and click Open. Recent projects can be accessed by navigating to Project > Recent projects and selecting the desired project from the list.

Once opened, project assets, such as ASN.1 schemas and generated source files, are visible in the Project pane.

Saving a Project

To save a project, navigate to Project > Save Project or Project > Save Project As.

Editing Schemas

The central area of the ACGUI window is dedicated to editing ASN.1 schema definition files.

Note

Additional information on the schema Editor can be found in the Interface topic of this guide.

```

1 - Module UsefulDefinitions (X.501:08/1997)
2 UsefulDefinitions {joint-iso-itu-t ds(5) module(1) usefulDefinitions(0) 3}
3 DEFINITIONS ::=
4 BEGIN
5
6 - EXPORTS All -
7 - The types and values defined in this module are exported for use in the other ASN.1 modules contained
8 - within the Directory Specifications, and for the use of other applications which will use them to access
9 - Directory services. Other applications may use them for their own purposes, but this will not constrain
10 - extensions and modifications needed to maintain or improve the Directory service.
11 ID ::= OBJECT IDENTIFIER
12
13 ds ID ::= {joint-iso-itu-t ds(5)}
14
15 - categories of information object
16 module ID ::= {ds 1}
17
18 serviceElement ID ::= {ds 2}
19
20 applicationContext ID ::= {ds 3}
21
22 attributeType ID ::= {ds 4}
23
24 attributeSyntax ID ::= {ds 5}
25
26 objectClass ID ::= {ds 6}
27
28 - attributeSet ID ::= {ds 7}
29 algorithm ID ::= {ds 8}
30
31 abstractSyntax ID ::= {ds 9}
32
33 - object ID ::= {ds 10}
34 - port ID ::= {ds 11}
35 dsaOperationalAttribute ID ::=

```

Creating a New Schema File

To create a new schema file, click New in the toolbar or navigate to File > New Schema File. A tab titled 'Untitled1.asn' is displayed in the central editing area of the GUI.

Define the schema by copy/pasting text or by manually entering text between the "DEFINITIONS ::= BEGIN" and "END" statements in the file. When finished, click Save file in the toolbar or navigate to File > Save file or File > Save file as. Click Validate to check for proper syntax and to confirm that no errors are present. Upon validation, a success or failure message is returned in the Error Log.

Editing a Schema File

To open an existing schema file for edit, click Open in the toolbar or navigate to File > Open File. The file is added to the project and is displayed as a tab in the central editing area. A schema file that is already included in a project can be opened by selecting the file name from the list of Schema/ASN.1 files in the Project pane.

At any point during editing, the schema can be saved and validated.

Deleting a Schema File

To remove a schema file from a project, right-click on the file name from the list of Schema/ASN.1 files in the Project pane and select Remove. Or, simply select the file within the Schema/ASN.1 files list and click Remove in the toolbar.

Compiling

Once a project is created and schemas are added, the schemas may be compiled to generate source code and related files.

Note

A target language must be selected for the project prior to compiling. To set the target language, navigate to Project > Project Settings > Output tab, and make a selection from the Application Language Type section.

Click Compile from the toolbar or navigate to Tools > Compile. Upon compilation, a success or failure message is returned in the Error Log.

Note

Upon clicking Compile, if any files have unsaved changes, a dialog box is displayed to prompt the user to save the files.

After compilation, changes can continue to be made to the schema and to the project settings. Recompilation can be done as needed.

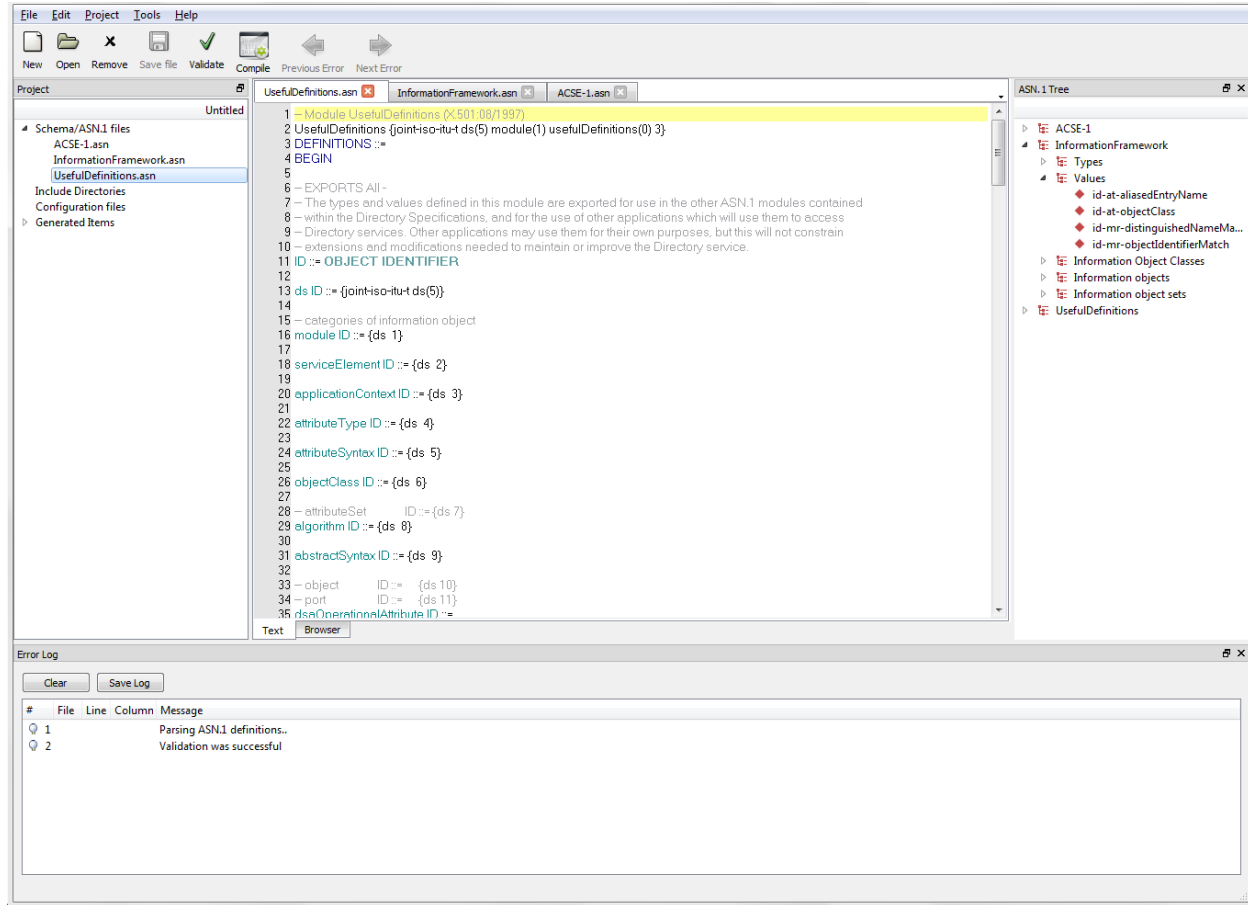
On Mac systems if Go code is being generated, sometimes an error message is displayed saying the "go mod init" command failed. This error message is caused by the location of the "go" command not being in the PATH that is used for GUI applications (which can be different from the PATH used for command shell sessions). This problem can be fixed by executing the following command (as an example) from a Terminal session on the Mac and then rebooting:

```
sudo launchctl config user path /usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

For this example /usr/local/bin is the location of the "go" command, and the other directories are the other locations that need to be in the PATH. The command will need to be adjusted accordingly for specific Mac systems.

Interface

The ACGUI interface is comprised of five parts: Editor, Project window, ASN.1 Tree window, Error Log window, and Project Settings.



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 or open a schema file. A new schema file can be opened by clicking New in the toolbar or by navigating to File > New Schema File. An existing schema file can be accessed via the Open button in the toolbar or navigating to File > Open File. The file is added to the current project and shown in the Editor.

```

1 - Module UsefulDefinitions (X.501:08/1997)
2 UsefulDefinitions {joint-iso-itu-t ds(5) module(1) usefulDefinitions(0) 3}
3 DEFINITIONS ::=
4 BEGIN
5
6 - EXPORTS All -
7 - The types and values defined in this module are exported for use in the other ASN.1 modules contained
8 - within the Directory Specifications, and for the use of other applications which will use them to access
9 - Directory services. Other applications may use them for their own purposes, but this will not constrain
10 - extensions and modifications needed to maintain or improve the Directory service.
11 ID ::= OBJECT IDENTIFIER
12
13 ds ID ::= {joint-iso-itu-t ds(5)}
14
15 - categories of information object
16 module ID ::= {ds 1}
17
18 serviceElement ID ::= {ds 2}
19
20 applicationContext ID ::= {ds 3}
21
22 attributeType ID ::= {ds 4}
23
24 attributeSyntax ID ::= {ds 5}
25
26 objectClass ID ::= {ds 6}
27
28 - attributeSet      ID ::= {ds 7}
29 algorithm ID ::= {ds 8}
30
31 abstractSyntax ID ::= {ds 9}
32
33 - object          ID ::= {ds 10}
34 - port            ID ::= {ds 11}
35 dsaOperationalAttribute ID ::=

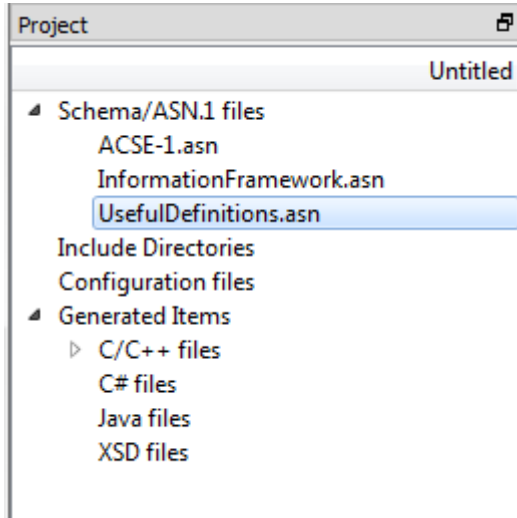
```

The Editor window is also used to display a schema browser for navigating within a validated schema. To display the browser after validating a schema, click on an item in the ASN.1 Tree window. The browser displays a hyperlinked version of the schema, centered on the definition of the selected item. Click the names of other defined types in the browser to show their definitions.

By default, schema documents are displayed as tabs in the Editor. The Text and Browser tabs at the bottom of the window are for schema editing and hyperlinked schema browsing, respectively. Schema files currently open for edit are displayed as tabs at the top of the Text tab.

Project Window

The Project window allows the user to interact with project assets, and contains the following components: Schema/ASN.1 files, Include Directories, Configuration files and Generated Items.



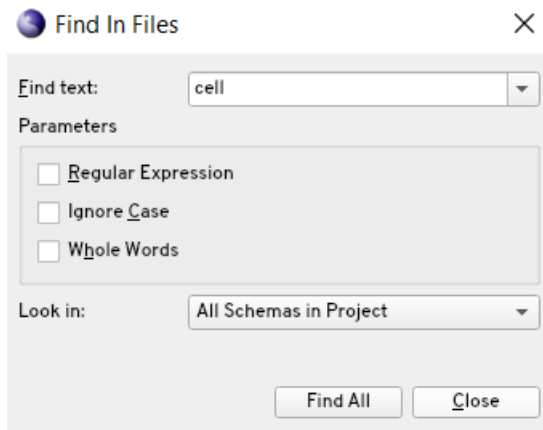
The Project window contains the following sections:

Schema/ASN.1 files	This section lists the files that contain the current project's ASN.1 schema definitions.
Include Directories	This section lists the directories containing auxiliary ASN.1 schema files. The current project's schema may import definitions from modules defined in an included directory.
Configuration file	This section lists the ASN1C compiler configuration file currently in use.
Generated Items	This section lists the files generated by the compiler, separated by target language.

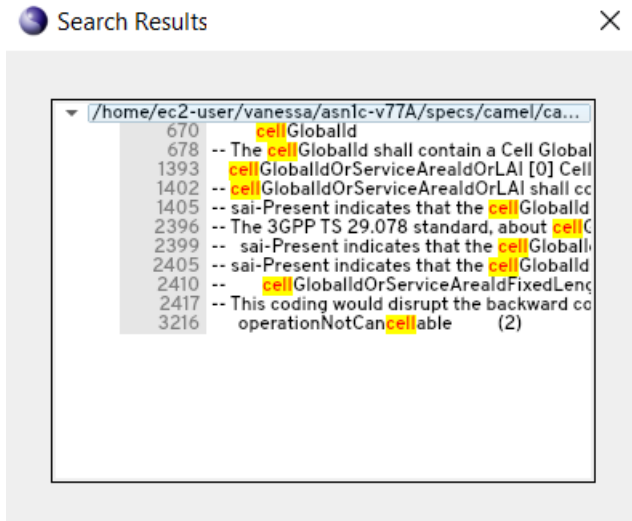
Click on a schema or configuration file in the Project window to open that file in the Editor. Right-click on any schema file, include directory, or configuration file to easily add another or remove that particular asset from the project.

Right-clicking on the Schema/ASN.1 files heading also displays options for 'Find in Files' and 'Replace in Files.'

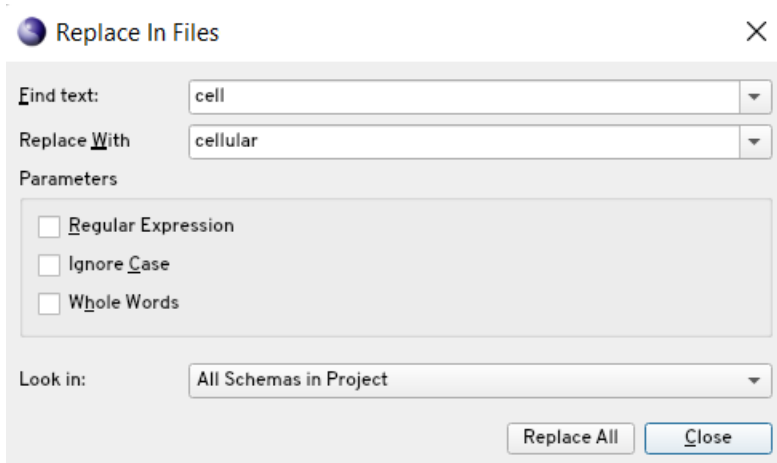
Selecting 'Find in Files' displays the Find in Files window, which provides users with the ability to enter any text to be located in either all schemas in the project, or only the schemas currently open in the editor. Once the Find parameters have been defined, click Find All to be presented with a list of files that contain the search text.



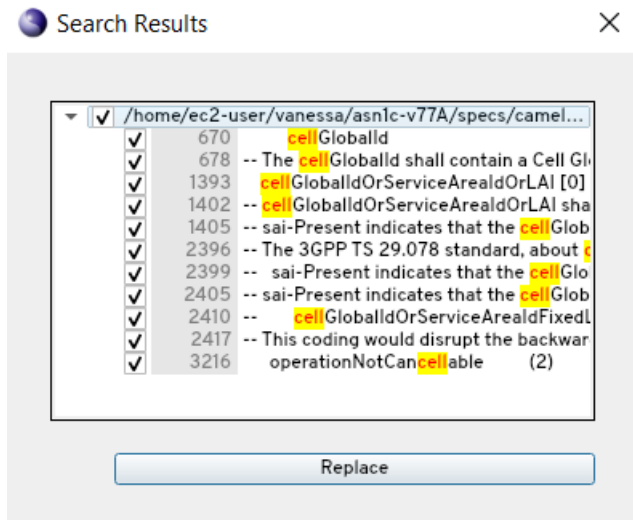
Click an entry within the results list to display the text in the editor.



Selecting 'Replace in Files' displays the Replace in Files window, which provides users with the ability to enter any text to be located and replaced in either all schemas in the project, or only the schemas currently open in the editor. Once the Find and Replace parameters have been defined, click Replace All to be presented with a list of files that include the word to be changed.

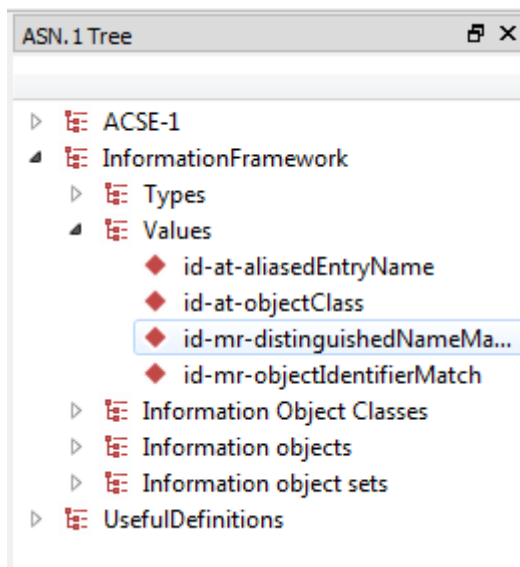


Click an entry within the results list to display the text in the editor, and if desired, deselect any check box next to an entry in the results list to prevent the replacement from happening on that result. Click Replace to perform the word replacement.



ASN.1 Tree Window

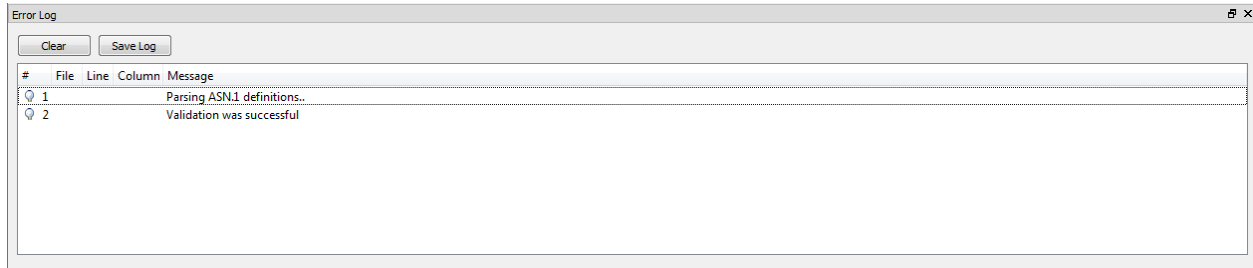
Once a schema has been validated or compiled in ACGUI, the ASN.1 Tree window provides an interactive view of the ASN.1 types defined in it.



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 within. Click on any node of the tree to 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 reports a success. If an error occurs, an error message is displayed.



In many cases, an error is associated with a particular portion of the schema being compiled. Click on an error to open the schema editor at the location in which the error occurred. If more than one error is reported, clicking Next Error and Previous Error in the toolbar moves the Editor window to the part of the schema where the next or previous error occurred.

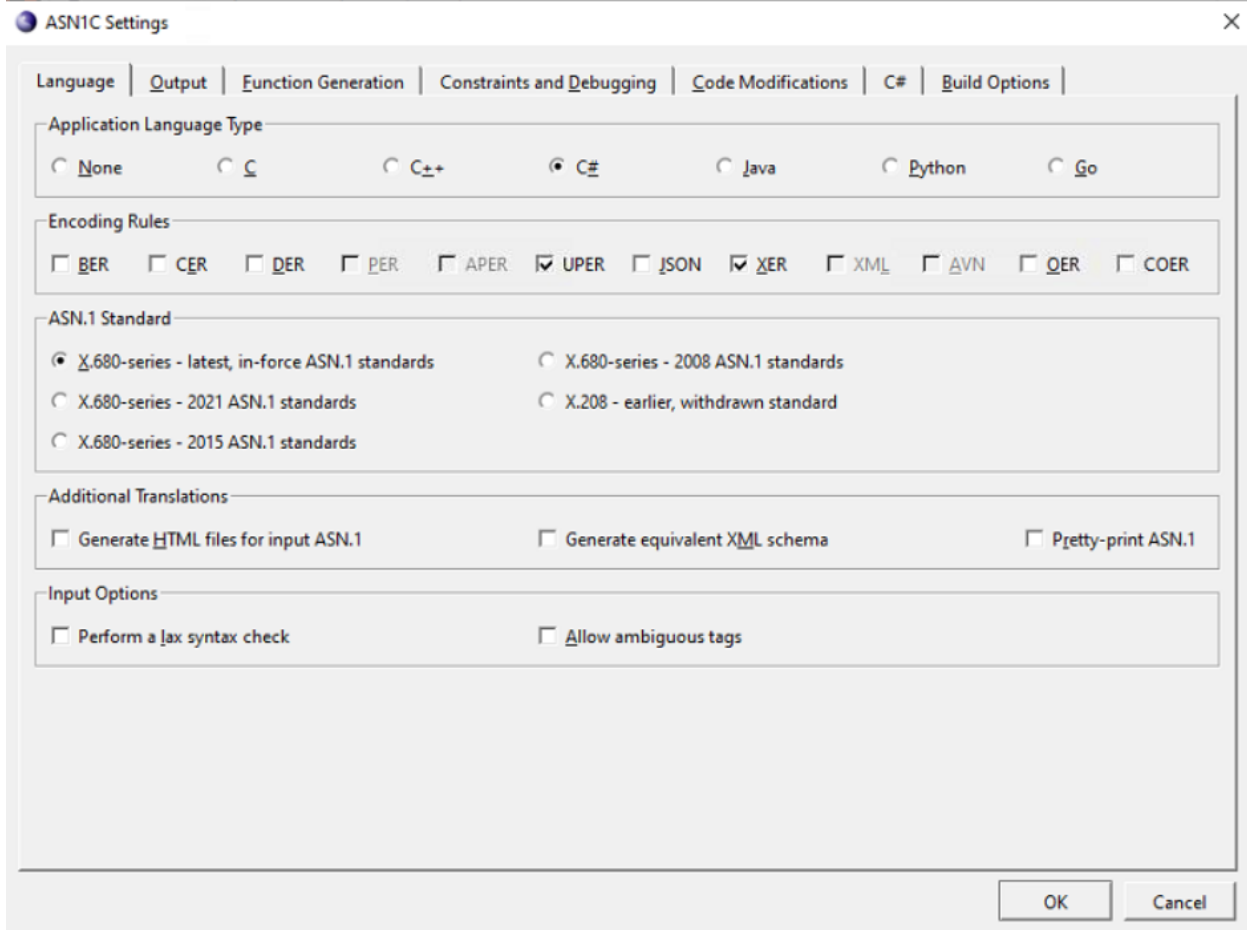
When the reported errors are no longer needed, click Clear in the Error Log window to remove them from display.

Project Settings

The Project Settings window is where details regarding the project, such as encoding rules, target language, and code features to generate are modified. The window contains the following tabs: Language, Output, Function Generation, Constraints and Debugging, Code Modifications, language-specific code modifications, and Build Options.

Language tab

The Language tab contains options for selecting a target language, encoding rules, ASN.1 Standards, translation, and input options.



The Language tab contains the following sections:

Application Language Type

This section provides users with the ability to define the target language for the project. A target language must be selected in order to compile a schema.

Depending on the target language selected, additional options are displayed within the Output tab.

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

C/C++ Output Options

Output code to .c/.h files based on module names

Output all code to a single .c/.h file

Output each generated function to its own source file

Default integer type: int32

Max lines per file:

Object directory:

Libraries Directory:

Binaries Directory:

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

C# Code Organization

Output code to directories based on module names

Code generation option: One .cs file per type File name:

Binaries Directory:

Libraries Directory:

For Java, the Java Code Organization section controls how generated code should be organized into directories based on the ASN.1 module for which they were generated. Alternatively, generated files are placed directly into the output directory.

Java Code Organization

Output code to directories based on module names

Class directory:

Encoding Rules

This section provides users with the ability to define one or more encoding rule sets to be selected for the generated code.

ASN.1 Standard

This section provides users with the ability to apply current or previously-defined ASN.1 standards to the generated code.

Additional Translations

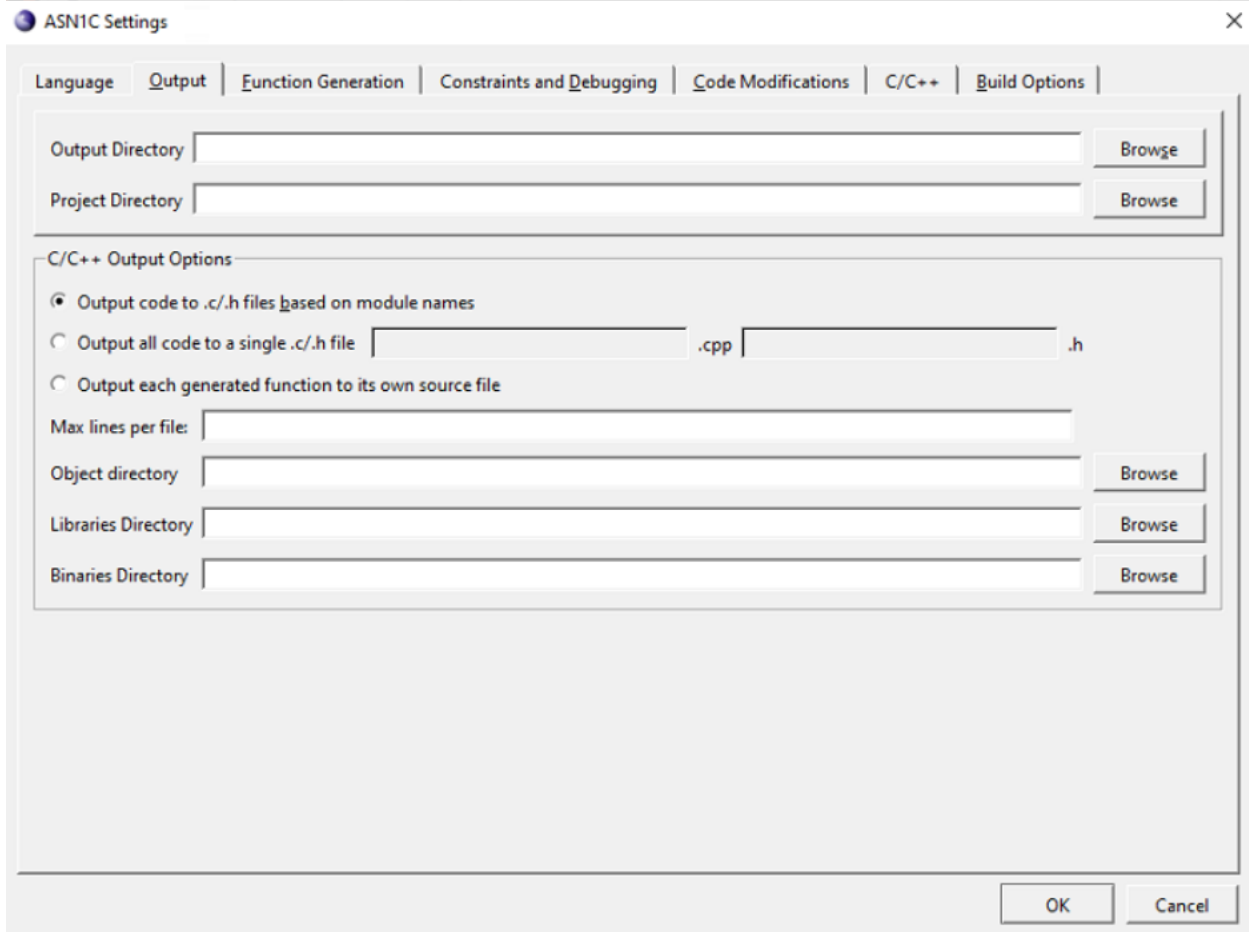
This section provides users with the ability to define the options for generating transformed versions of the input schema, such as HTML or pretty-printed.

Input Options

This section provides users with the ability to define how strict the compiler is when parsing ASN.1 schema.

Output tab

The Output tab contains options for selecting encoding rules, as well as output and project directories.

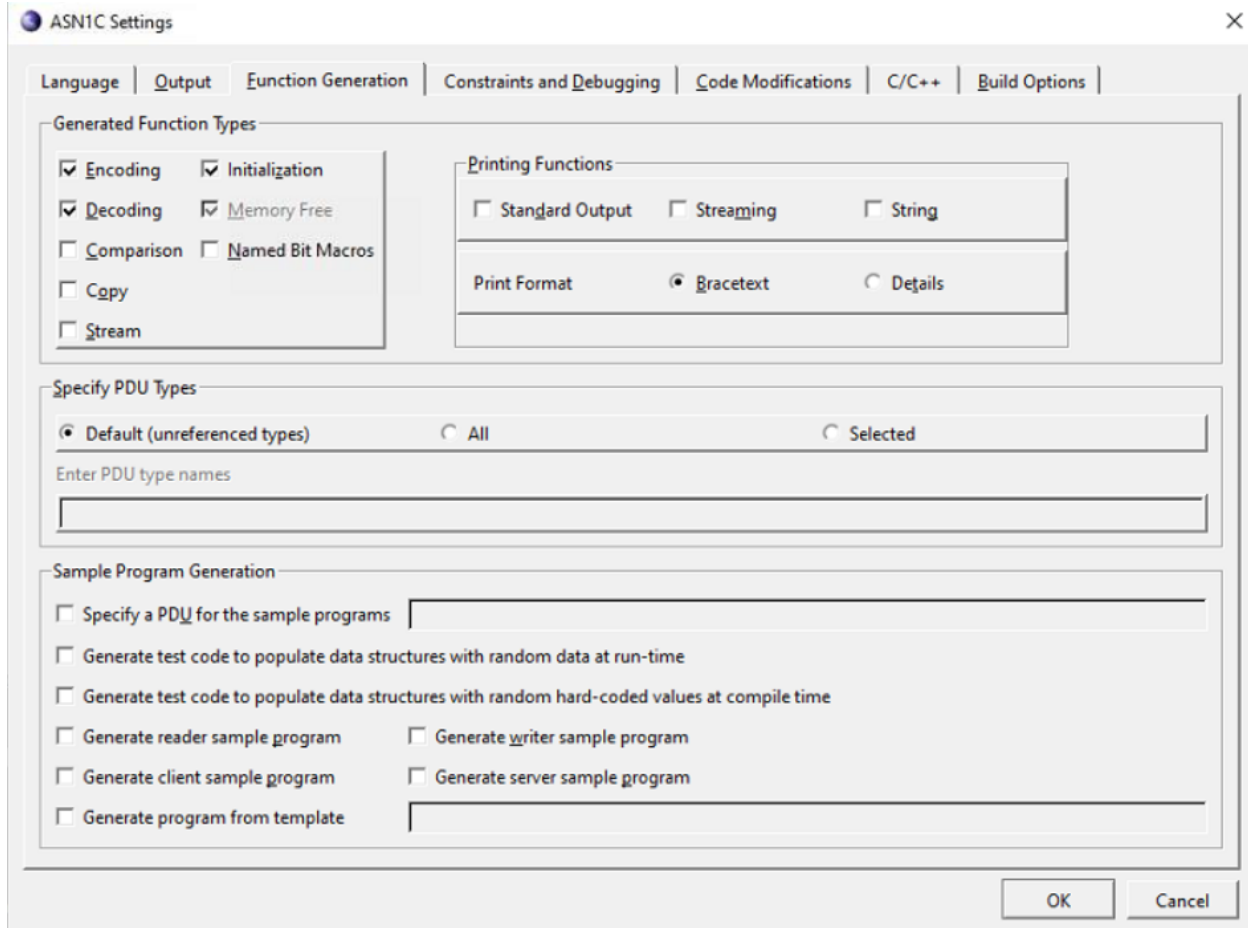


The Output tab contains the following sections:

- | | |
|--|---|
| Output Directory and Project Directory | This section provides users with the ability to define in which directory the code output and/or project should be stored. |
| Language-specific Output Options | This section provides users with the ability to define additional output options specific to the language type chosen on the Language tab, as well as the ability to define additional directories. |

Function Generation tab

The Function Generation tab provides settings for the functionality to include in generated code.



The Function Generation tab contains the following sections:

Generated Function Types	This section provides users with the ability to define granular control of which 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.
Specify PDU Types	This section provides users with the ability to define which productions to select as PDUs.
Sample Program Generation	This section provides users with the ability to define the generation of simple encoding and decoding programs, which demonstrate using the generated code. Additionally, the sample writer program can optionally encode randomly-generated test data.
Language-specific Functions	Depending on the target language selected, additional options may be displayed within this section of the Function Generation tab. 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, but may be turned off.

C/C++ Generated Functions

- Generate Initialization Functions
- Generate Memory Free Functions
- Generate Named Bit Macros

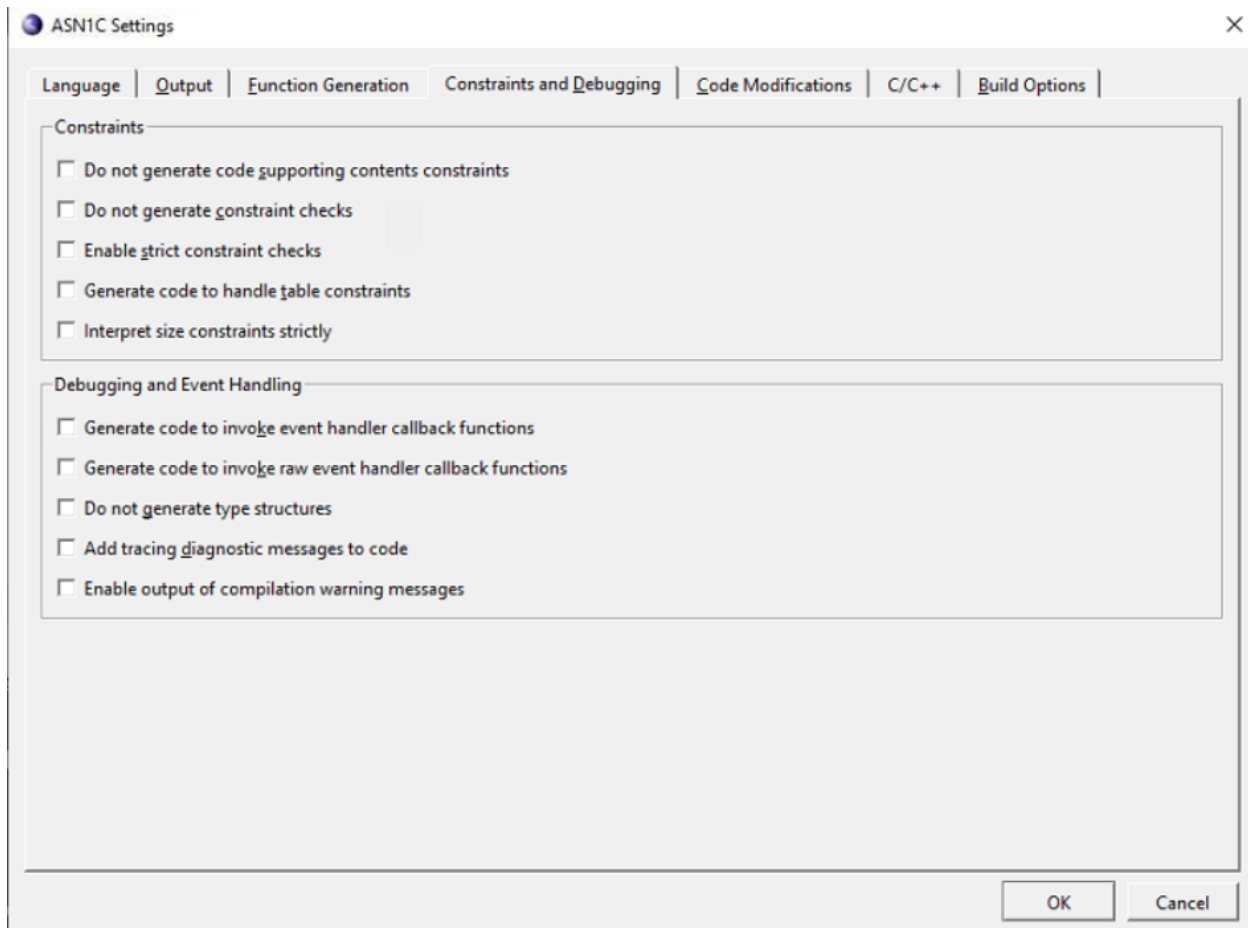
For Java, get and set methods can be generated for members of generated classes. It is also possible to generate methods that can fetch certain types of metadata (for example, if an element is optional). A similar option exists for C#.

Java Function Options

- Generate getter and setter methods
- Generate metadata methods

Constraints and Debugging tab

The Constraints and Debugging tab holds settings related to constraint handling, event handling, and logging in generated code.



The Constraints and Debugging tab contains the following sections:

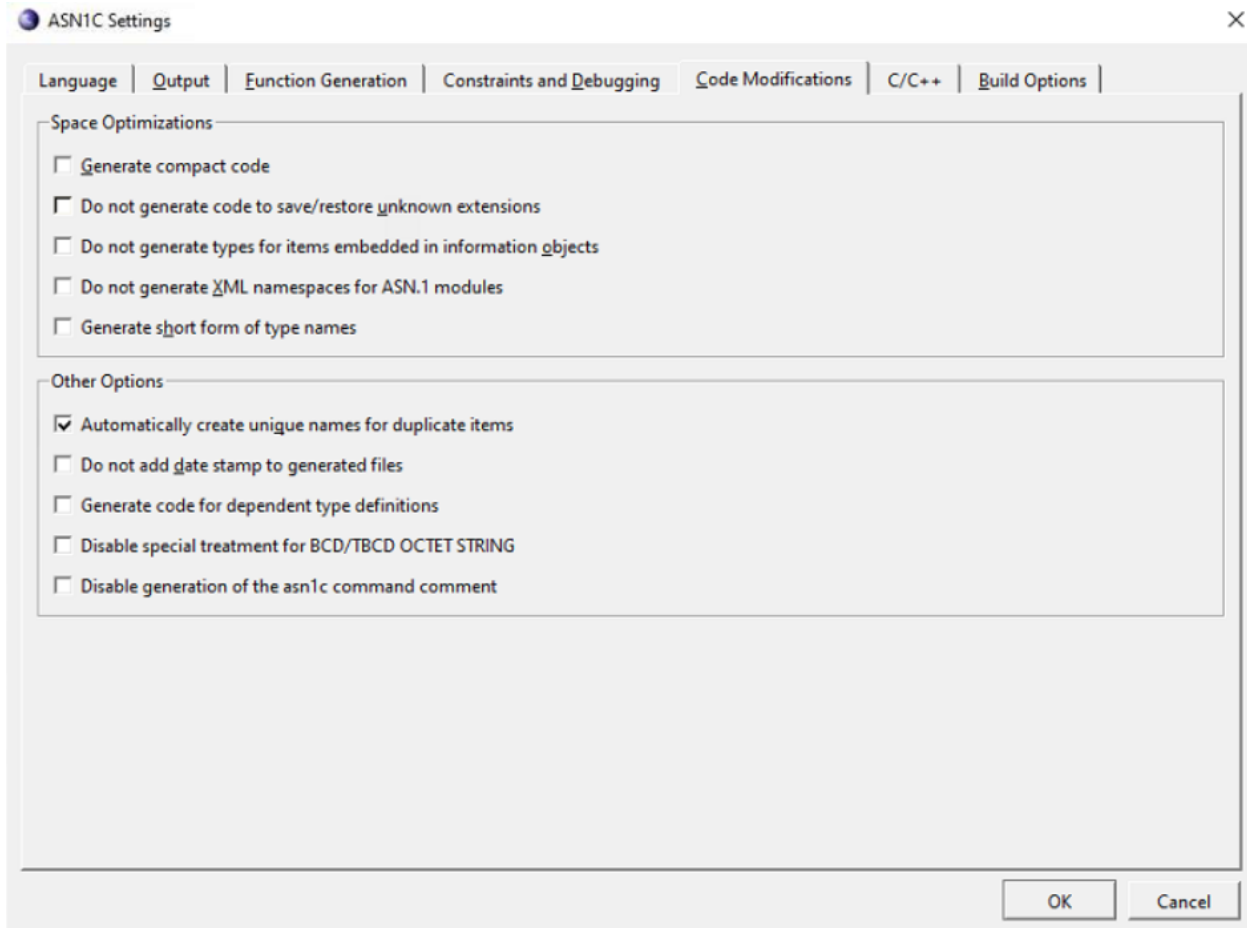
Constraints

This section provides users with the ability to add or remove various types of restriction checks from the generated code.

Debugging and Event Handling This section provides users with the ability to add debug tracing and event hooks. In addition to enabling event callbacks, generation of type structures can also be disabled, in which case generated decode functionality simply calls user-created event handlers and does not perform its own decoding operation.

Code Modifications tab

The Code Modifications tab contains a number of options for generating simplified code.



The Code Modifications tab contains the following sections:

Space Optimizations This section provides users with the ability to remove unwanted or unneeded functionality and shorten the names of generated types.

Other Options This section provides users with the ability to define several miscellaneous settings, including the option to generate code for types that have been imported into the current schema.

Language-specific tab

Additional code modification options that are language-specific are shown in a separate tab next to the Code Modifications tab. The label and contents of this tab changes based on the language selected within the Output tab.

For C/C++, the tab is displayed as follows, and includes several settings for adjusting how ASN.1 types are mapped to native C/C++ types:

C/C++ Code Modifications

Add a header guard prefix

Add a C++ namespace

SEQUENCE OF/SET OF use: default linked list dynamic array static array

Generate static member variables in choice constructs

Use enumerated types instead of integers

Generate fully qualified enumerated constants

Disable fixed sizing for small bit strings

Support indefinite PER lengths

Pad PER BIT STRING-contained values

Generate code compatible with 64-bit architectures

Generate code that uses C++11 features including STL

For C#, the tab is displayed as follows, and includes settings to allow for manipulating the namespace into which code is generated:

Note

The Java tab contains similar options.

C# Code Modifications

Specify namespace

Specify namespace prefix

Build Options tab

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

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

<input type="checkbox"/> Generate Visual Studio Project <input type="radio"/> VS 2019 <input checked="" type="radio"/> VS 2017 <input type="radio"/> VS 2015 <input type="radio"/> VS 2013 <input type="radio"/> VS 2012 <input type="radio"/> VS 2010 <input type="radio"/> VS 2008 <input type="radio"/> VS 2005 <input type="radio"/> VS 2003 <input type="radio"/> VS 6.0 (1998)	<input type="checkbox"/> Generate Makefiles <input type="radio"/> Windows (<u>n</u> make) <input checked="" type="radio"/> GNU
<input type="checkbox"/> Build Libraries <input type="radio"/> Generate <u>s</u> tatic libraries <input type="radio"/> Generate <u>s</u> hared libraries <input type="checkbox"/> Generate <u>m</u> ulti-threaded libraries	<input type="checkbox"/> C/C++ Compile Optimization <input checked="" type="radio"/> Default <input type="radio"/> Space Optimization <input type="radio"/> Speed (Time) Optimization

Link applications using shared libraries
 Link applications against 64-bit libraries

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

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

For C#, the tab is as follows:

Build Options

Generate a list of .cs files (in <modulename>.mk)

Generate a makefile

Generate a Visual Studio Project

- VS 2019
- VS 2017
- VS 2015
- VS 2013
- VS 2012
- VS 2010
- VS 2008
- VS 2005

Strongly named key file:

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 is also available.

For Java, the tab is as follows:

Build Options

Generate a list of .java files (in <modulename>.mk)

Generate an Ant build script

Generate a batch file or shell script

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

For Python, the tab is as follows:

Build Options

Create a batch file or shell script to generate Python code

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

For Go, the tab is as follows:

Build Options

- Create a makefile to generate Go code
- Create a JSON file with random test data
- Don't create a main.go file

The Go code generator can create a makefile to generate and build Go code. The generator can also create a JSON file with random test data. Additionally, an option for the generator not to create a main.go file (for instance, if there already is one that has been modified) is available.

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. An 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 definitions 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 corresponding 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, implen=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, implen=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, implen=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, implen=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, implen=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 this 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 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 this case encodes the instance attributes which correspond to the ASN.1 elements defined 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 *AsnIBerEncodeBuffer* 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 *AsnIBerEncodeBuffer* 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* argument 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 *AsnIBitString* 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:

```
enobuf = 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 = enobuf.encode_integer(10)
```

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

The procedure for invoking a static method is similar. The general form is `<classname>.encode(value, enobuf, 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, enobuf)
```

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(enobuf)
```

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

enobuf = 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 their case decodes data into the instance attributes which correspond to the ASN.1 elements defined 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 object 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 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, implicit). 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 defined 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 be 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 = AsnlJsonDecodeBuffer(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 defined 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 Python 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 specified on 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):
```