ASN1C

ASN.1 Compiler User's Guide for Go

Version 7.6

Objective Systems, Inc.

January 2022

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

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

Table of Contents

Chapter 1. Overview of ASN1C for Go

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 Go 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 Python Compiler User's Manual* : Python code generation

Each ASN.1 module that is encountered in an ASN.1 schema source file results in the generation of a Go source file. The name of the file is the ASN.1 module name, with hyphens removed and changed to camel-case, and an extension of '.go'.

There is also a set of functions that form the run-time component of the Go package. These functions provide 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 Go 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:

- Packed Encoding Rules (PER) as published in the ITU-T X.691 and ISO/IEC 8825-2 standards. Includes both the aligned and unaligned variants (note: the unaligned variant is commonly known as UPER).
- JSON Encoding Rules (JER) as published in the ITU-T X.697 and ISO/IEC 8825-8 standards.

ASN1C for Go 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 ITU-T X.681 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 Go.

ASN1C Go Command Line Options

The following table shows a summary of the command line options that have meaning when generating Go code:

Option	Argument	Description
-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 *Compiler Configuration File* section.
-depends	None	Without this option, the compiler only generates Go code for the files that were explicitly given as input, and not for any files it may automatically load on account of IMPORTS. This option tells the compiler to also generate code for files it automatically loads, but only that code which is necessary to satisfy dependencies that originate in the main file set.
-genPrint -print	None	Generate code to print object contents. The open source go-spew package (https://github.com/davecgh/go-spew) is used for this purpose.
-genTest -test	None	Generate code to populate a PDU object with random test data. This code will be embedded in the body of a generated main.go program.
-I	<directory>	This option specifies a directory where the compiler should 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	This option is used to generate encode/decode functions that implement

Option	Argument	Description
		the Javascript Object Notation (JSON) Encoding Rules (JER) as specified in the X.697 ASN.1 standard.
-lax	None	Suppress generation of code to check constraints.
-list	None	Generate listing. This will dump the source ASN.1 to the standard output device as it is parsed. This can be useful for finding parse errors.
-nodecode	None	Suppress generation of decode functions.
-noencode	None	Suppresses generation of encode functions.
-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.
-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>. 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).
-no-go-main	None	Suppress the generation of a main.go file. By default, this file is generated with options to encode (-writer) and decode (-reader) a message of a PDU type.
-o	<directory>	Specify the name of a directory to which all of the generated files will be written.

Option	Argument	Description
-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.
-per	None	This option is used to generate encode/decode functions that implement the Packed Encoding Rules (PER - aligned variant) as specified in the ASN.1 standards.
-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.
-uper	None	This option is used to generate encode/decode functions that implement the Packed Encoding Rules (PER - unaligned variant) as specified in the ASN.1 standards.
-warnings	None	Output information on compiler generated warnings.

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. Configuration items that are applicable to Go 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

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

Production Level

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

Name	**Values**	**Description**
<name> </name>	production name	This attribute identifies the production (type) to which this section applies. The name may also be specified as an attribute. In either case, it is required.
<typePrefix> </typePrefix>	prefix text	This is used to specify a prefix that will be applied to generated Go type names for this production. This can be used to prevent name clashes if multiple modules are involved in a compilation and they 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.
<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.

Compiler Error Reporting

Errors that can occur when generating source code from ASN.1 take two forms: syntax errors and semantic 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 names must begin with an uppercase letter, production (type) names 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, Go does not allow hyphens in identifiers and prefers names be in camel-case form. ASN1C will attempt to transform the ASN.1 names into the Go preferred format.

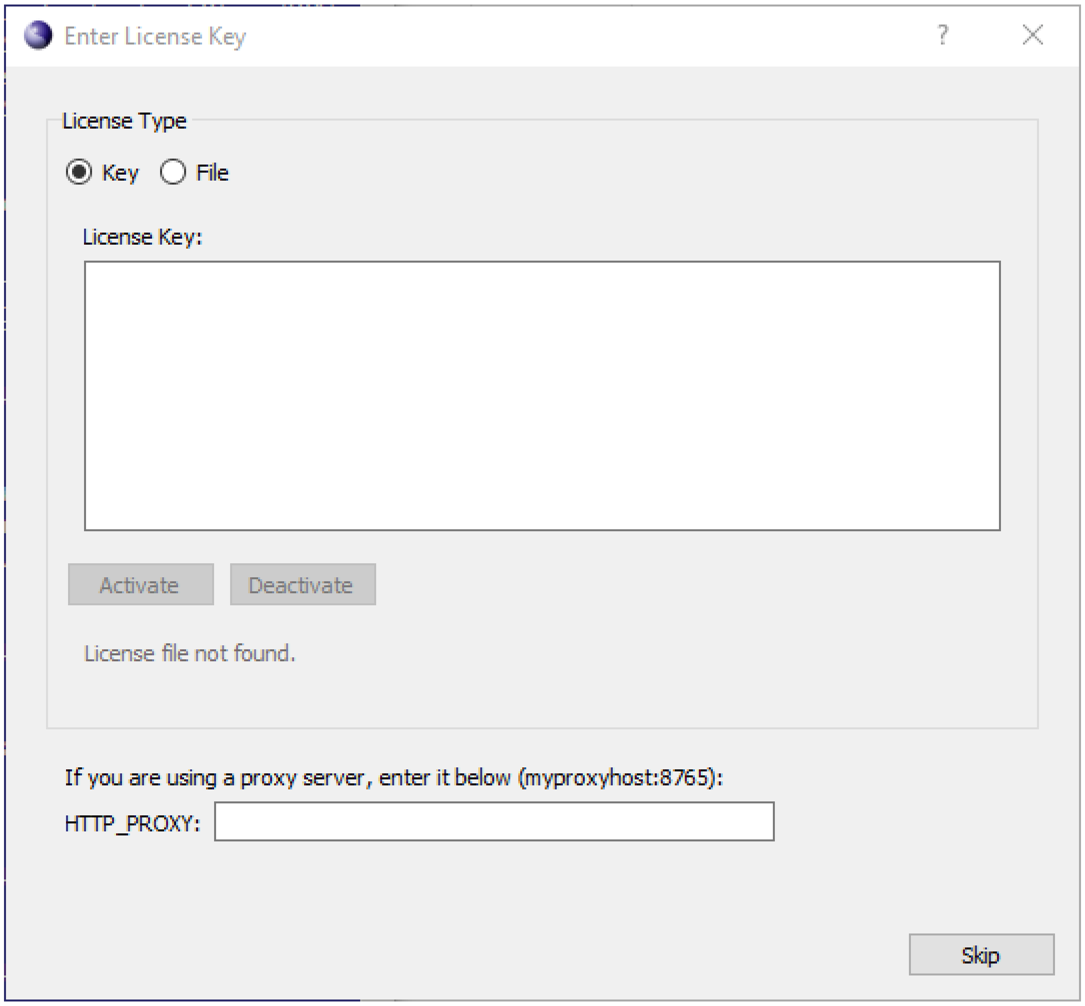
Semantic errors occur on the compiler back-end as the code is being generated. In this case, parsing was successful, but the compiler does not know how to generate the code. These errors are flagged by embedding error messages directly in the generated code.

Chapter 3. ASN1C GUI Users Guide

Quick Start

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

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



License file not found.

If you are using a proxy server, enter it below (myproxyhost:8765):

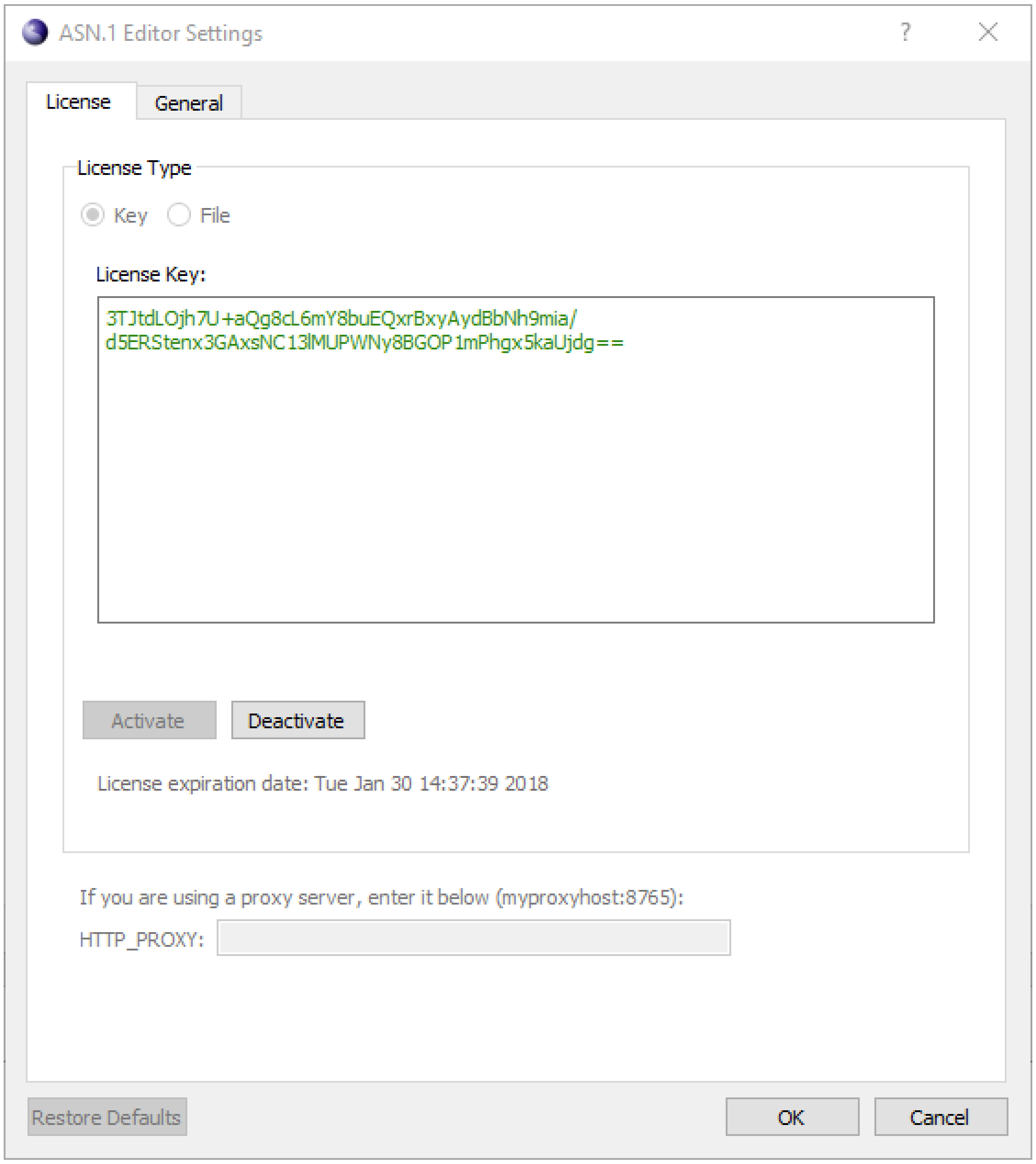
If you see this window, and it's not showing a current license key, you can right click in the text box and paste in your license key. Then click the Activate button to unlock ASN1C.

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

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

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

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



License Type

[✓ Key] File

License Key:

3TJtdLOjh7U+aQg8cL6mY8buEQxrBxyAydBbNh9mia/
d5ERStenx3GAxsNC13lMUPWNy8BGOP1mPhgx5kaUjdg==

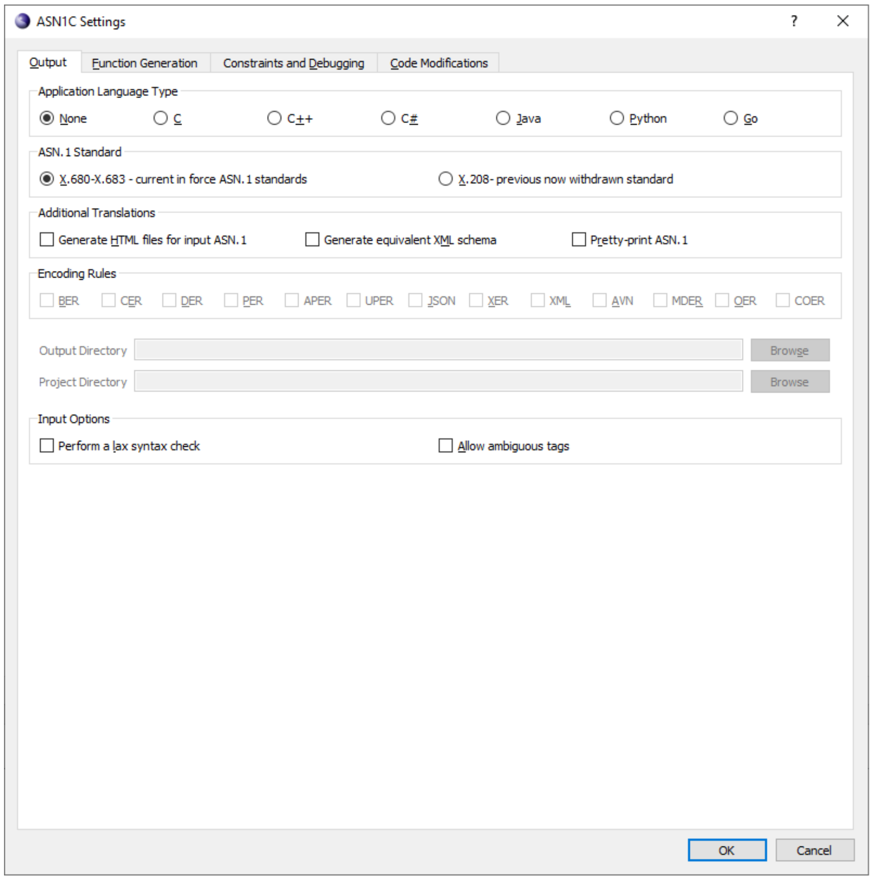
License expiration date: Tue Jan 30 14:37:39 2018

If you are using a proxy server, enter it below (myproxyhost:8765):

HTTP_PROXY:

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

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


ASN1C Settings

Output | Function Generation | Constraints and Debugging | Code Modifications

Application Language Type
(● None) ○ C ○ C++ ○ C# ○ Java ○ Python ○ Go

ASN.1 Standard
(● X.680-X.683 - current in force ASN.1 standards) ○ X.208- previous now withdrawn standard

Additional Translations
☐ Generate HTML files for input ASN.1 ☐ Generate equivalent XML schema ☐ Pretty-print ASN.1

Encoding Rules
☐ BER ☐ CER ☐ DER ☐ PER ☐ APER ☐ UPER ☐ JSON ☐ XER ☐ XML ☐ AVN ☐ MDER ☐ OER ☐ COER

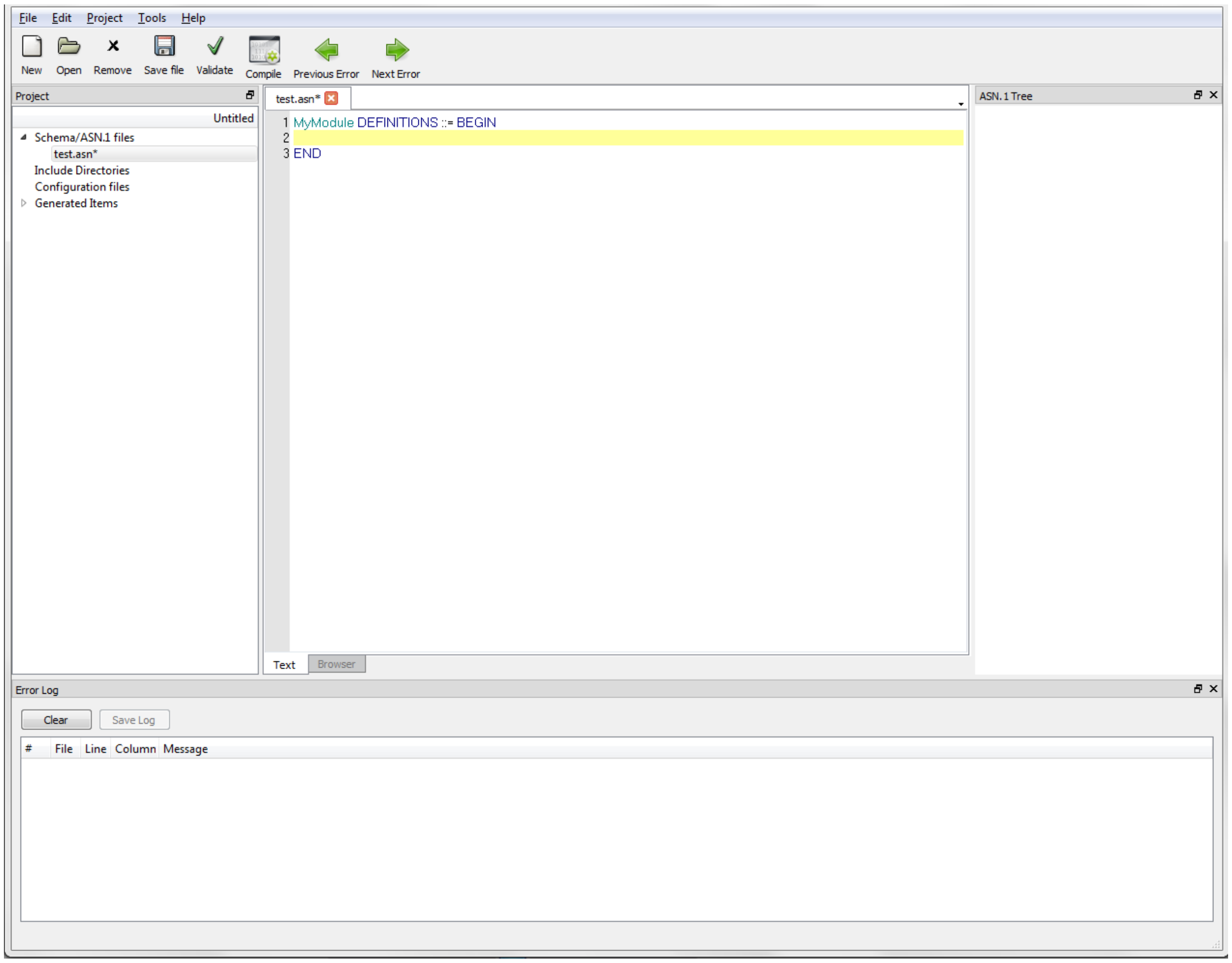
Output Directory [Browse]
Project Directory [Browse]

Input Options
☐ Perform a lax syntax check ☐ Allow ambiguous tags

OK Cancel

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

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



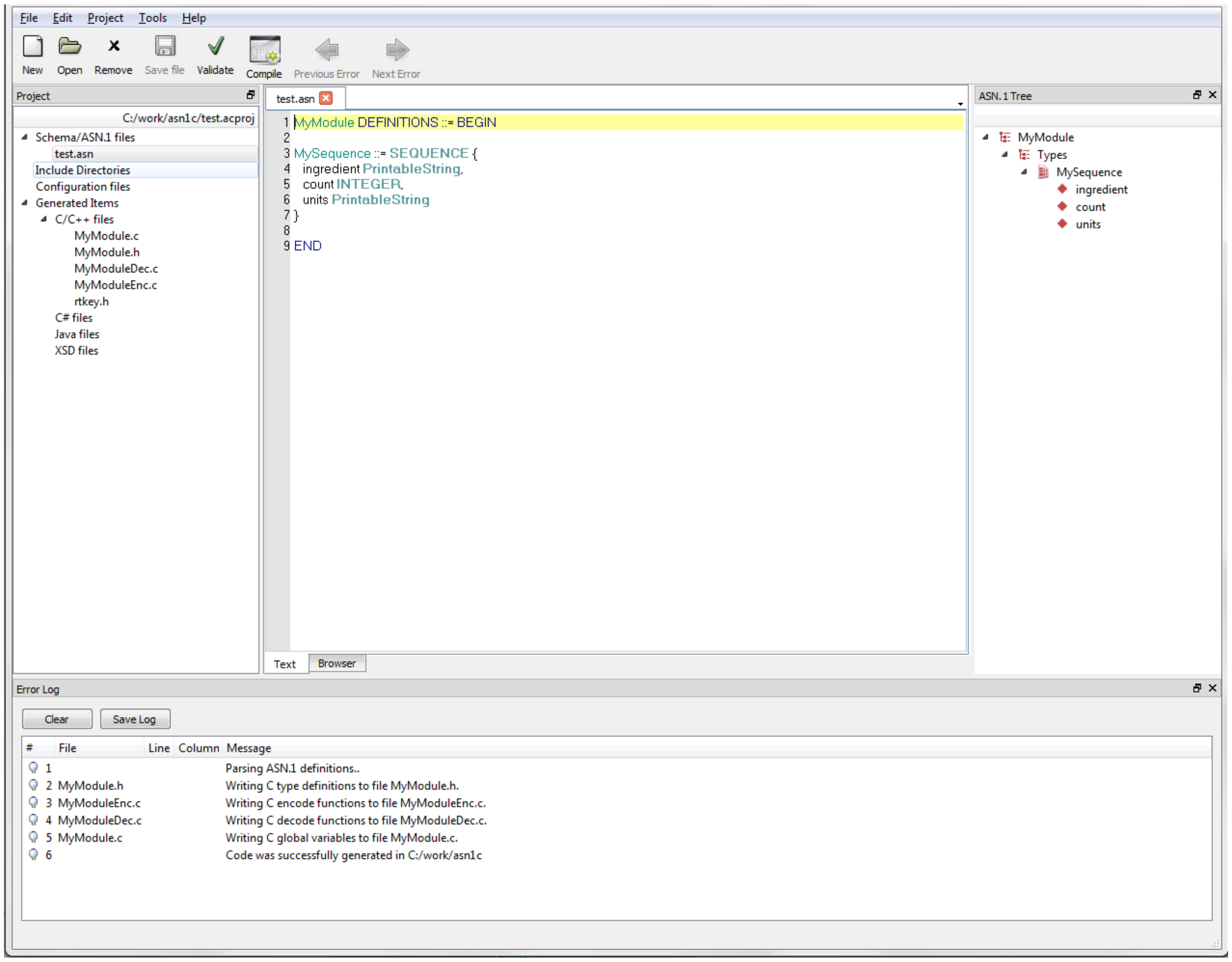
```
MyModule DEFINITIONS ::= BEGIN

END
```

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

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

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

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



```
MyModule DEFINITIONS ::= BEGIN

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

END
```

```
#  File             Line  Column  Message
1                                 Parsing ASN.1 definitions..
2  MyModule.h                     Writing C type definitions to file MyModule.h.
3  MyModuleEnc.c                  Writing C encode functions to file MyModuleEnc.c.
4  MyModuleDec.c                  Writing C decode functions to file MyModuleDec.c.
5  MyModule.c                     Writing C global variables to file MyModule.c.
6                                 Code was successfully generated in C:/work/asn1c
```

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

Creating a Project

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

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

Editing Schemas

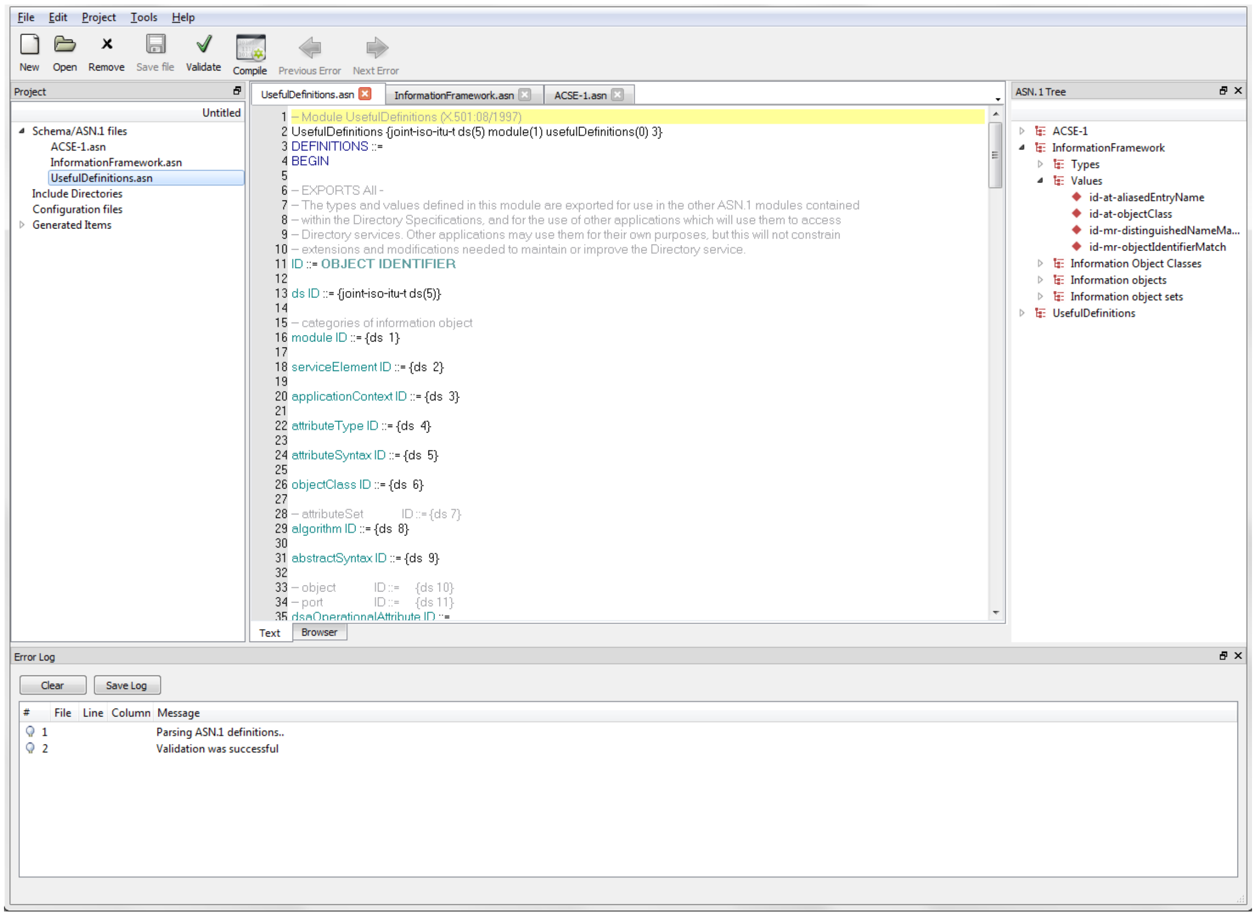
The central area of the ACGUI window is dedicated to editing ASN.1 schema definition files. To create a new file, click the "New" button in the toolbar or select "File->New Schema File" in the menus. To open an existing schema file, either click the "Open" button in the toolbar or select "File->Open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

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

Compiling

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

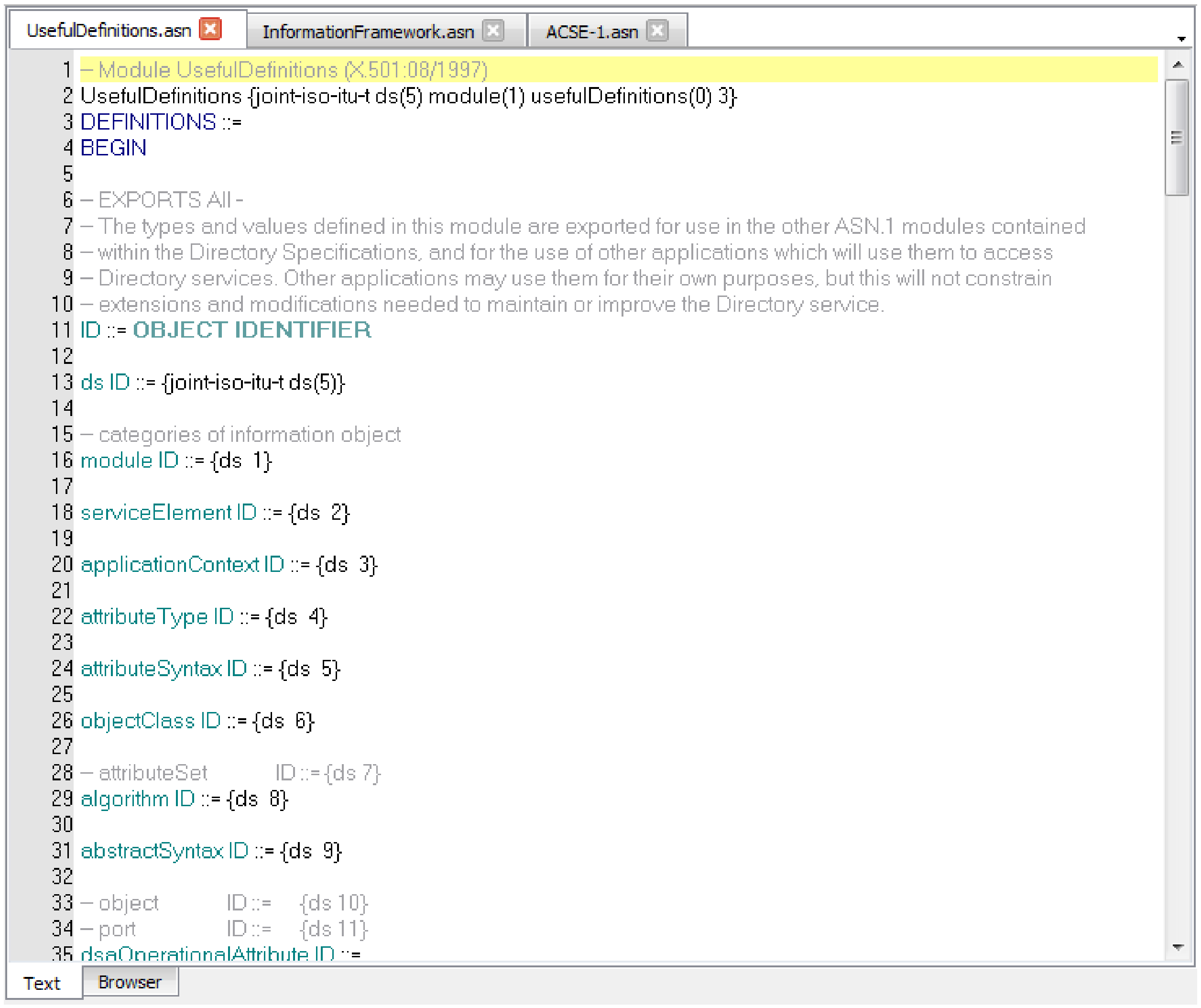
Interface



```
-- Module UsefulDefinitions (X.501:08/1997)
UsefulDefinitions {joint-iso-itu-t ds(5) module(1) usefulDefinitions(0) 3}
DEFINITIONS ::=
BEGIN

-- EXPORTS All -
-- The types and values defined in this module are exported for use in the other ASN.1 modules contained
-- within the Directory Specifications, and for the use of other applications which will use them to access
-- Directory services. Other applications may use them for their own purposes, but this will not constrain
-- extensions and modifications needed to maintain or improve the Directory service.
ID ::= OBJECT IDENTIFIER

ds ID ::= {joint-iso-itu-t ds(5)}

-- categories of information object
module ID ::= {ds  1}

serviceElement ID ::= {ds  2}

applicationContext ID ::= {ds  3}

attributeType ID ::= {ds  4}

attributeSyntax ID ::= {ds  5}

objectClass ID ::= {ds  6}

-- attributeSet            ID ::= {ds 7}
algorithm ID ::= {ds  8}

abstractSyntax ID ::= {ds  9}

-- object           ID ::=     {ds 10}
-- port             ID ::=     {ds 11}
dsaOperationalAttribute ID ::=
```

```
#   File  Line  Column  Message
1                       Parsing ASN.1 definitions..
2                       Validation was successful
```

Editor



```
– Module UsefulDefinitions (X.501:08/1997)
UsefulDefinitions {joint-iso-itu-t ds(5) module(1) usefulDefinitions(0) 3}
DEFINITIONS ::=
BEGIN

– EXPORTS All -
– The types and values defined in this module are exported for use in the other ASN.1 modules contained
– within the Directory Specifications, and for the use of other applications which will use them to access
– Directory services. Other applications may use them for their own purposes, but this will not constrain
– extensions and modifications needed to maintain or improve the Directory service.
ID ::= OBJECT IDENTIFIER

ds ID ::= {joint-iso-itu-t ds(5)}

– categories of information object
module ID ::= {ds  1}

serviceElement ID ::= {ds  2}

applicationContext ID ::= {ds  3}

attributeType ID ::= {ds  4}

attributeSyntax ID ::= {ds  5}

objectClass ID ::= {ds  6}

– attributeSet          ID ::= {ds 7}
algorithm ID ::= {ds  8}

abstractSyntax ID ::= {ds  9}

– object           ID ::=     {ds 10}
– port             ID ::=     {ds 11}
dsaOperationalAttribute ID ::=
```

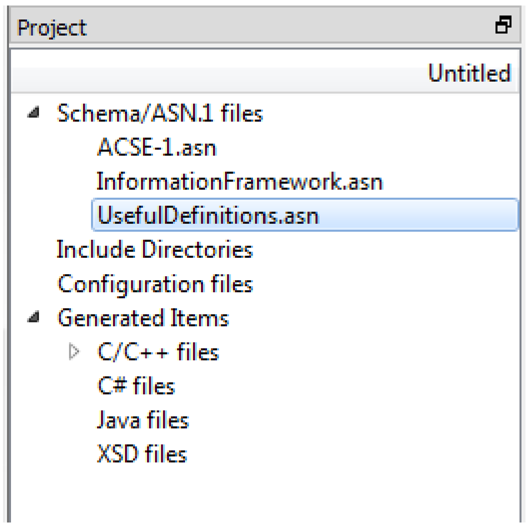
The central part of the ACGUI window is the schema editor. From here, schema files can be viewed and edited. To begin editing an ASN.1 schema, create a new file or open an existing file via the toolbar or menu. The file will be added to the current project and shown in the editor.

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

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

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

Project Window

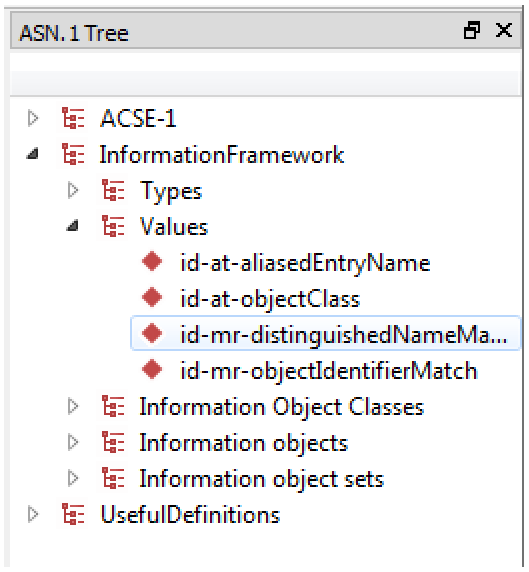


The project window allows the user to interact with project assets.

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

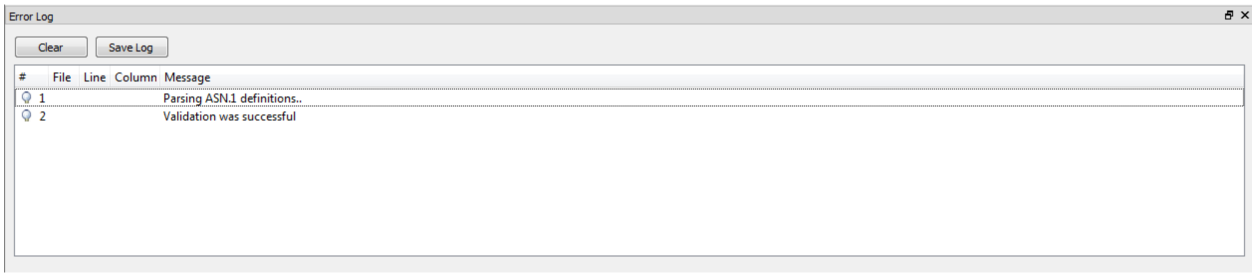
Clicking on a schema file or configuration file in the project window will open that file in the editor. A right-click context menu is also provided for schema files, include directories, and configuration file for adding or removing these assets from the project.

ASN.1 Tree Window



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

Error Log Window



#	File	Line	Column	Message
1				Parsing ASN.1 definitions..
2				Validation was successful

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

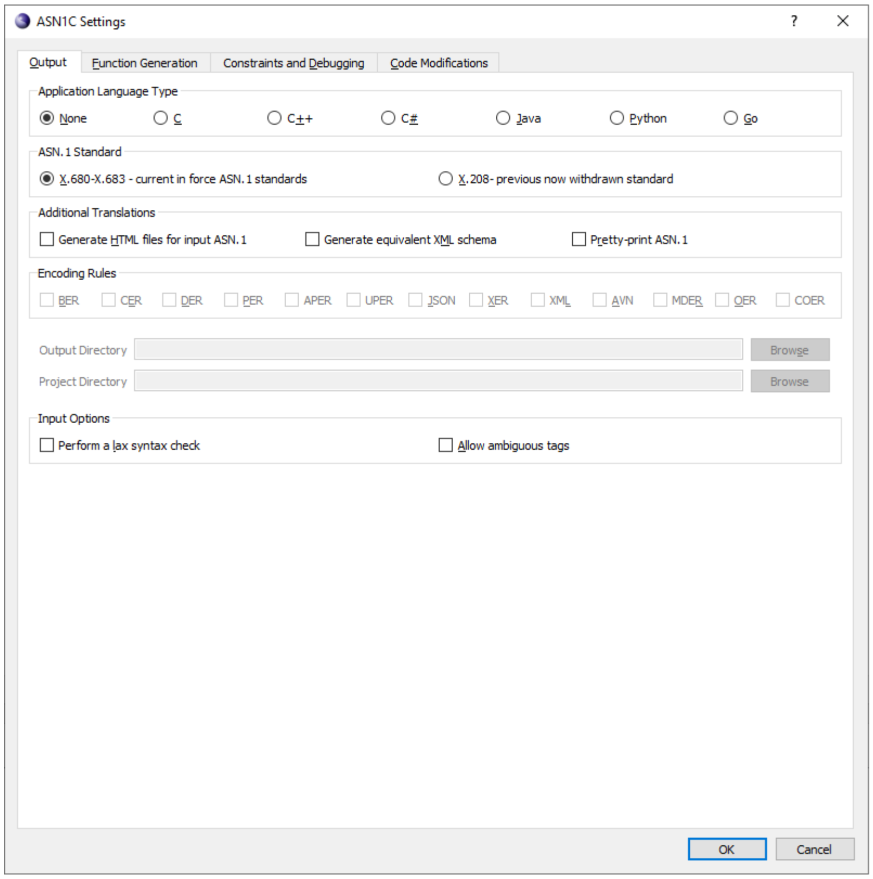
In many cases, an error will be associated with a particular portion of the schema being compiled. In these cases, clicking on an error will open the schema editor to the location that the error occurred. If more than one error is reported, clicking the "Next Error" and "Previous Error" buttons in the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

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

Project Settings

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

Output tab



ASN1C Settings

Output | Function Generation | Constraints and Debugging | Code Modifications

Application Language Type: [✓ None] C C++ C# Java Python Go

ASN.1 Standard: [✓ X.680-X.683 - current in force ASN.1 standards] X.208- previous now withdrawn standard

Additional Translations: ☐ Generate HTML files for input ASN.1 ☐ Generate equivalent XML schema ☐ Pretty-print ASN.1

Encoding Rules: ☐ BER ☐ CER ☐ DER ☐ PER ☐ APER ☐ UPER ☐ JSON ☐ XER ☐ XML ☐ AVN ☐ MDER ☐ OER ☐ COER

Output Directory: Browse

Project Directory: Browse

Input Options: ☐ Perform a lax syntax check ☐ Allow ambiguous tags

OK Cancel

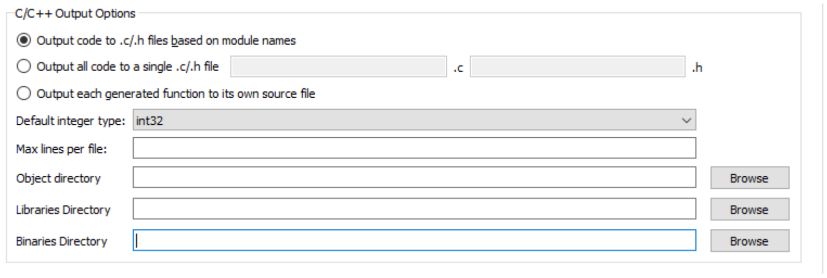
The "Output" tab contains options for selecting a target language, encoding rules, output directory. In order to compile a schema, a target language must be selected under "Application Language Type".

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

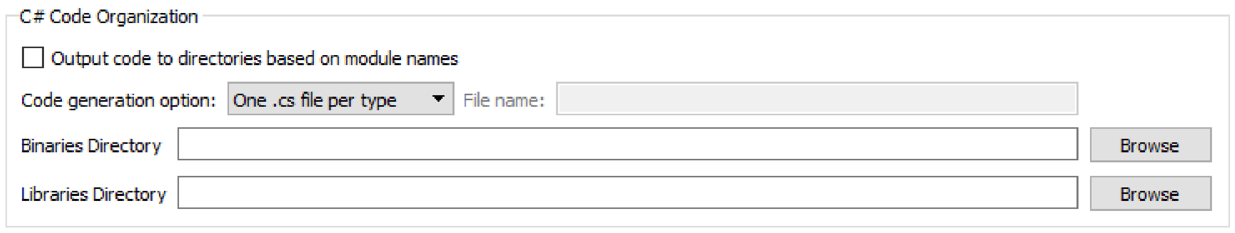
"Encoding Rules" allows for one or more encoding rule set to be selected for generated code.

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

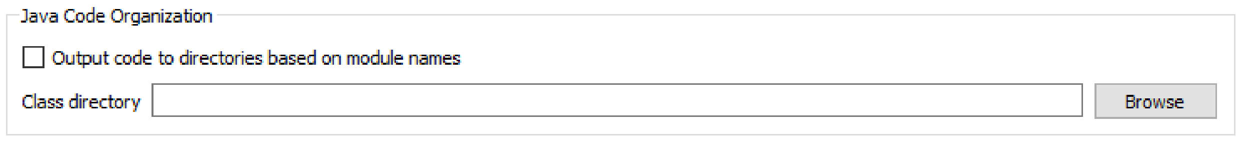
Depending on the target language selected, additional options are shown.


C/C++ Output Options
● Output code to .c/.h files based on module names
○ Output all code to a single .c/.h file
.c
.h
○ Output each generated function to its own source file
Default integer type: int32
Max lines per file:
Object directory
Libraries Directory
Binaries Directory
Browse
Browse
Browse

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

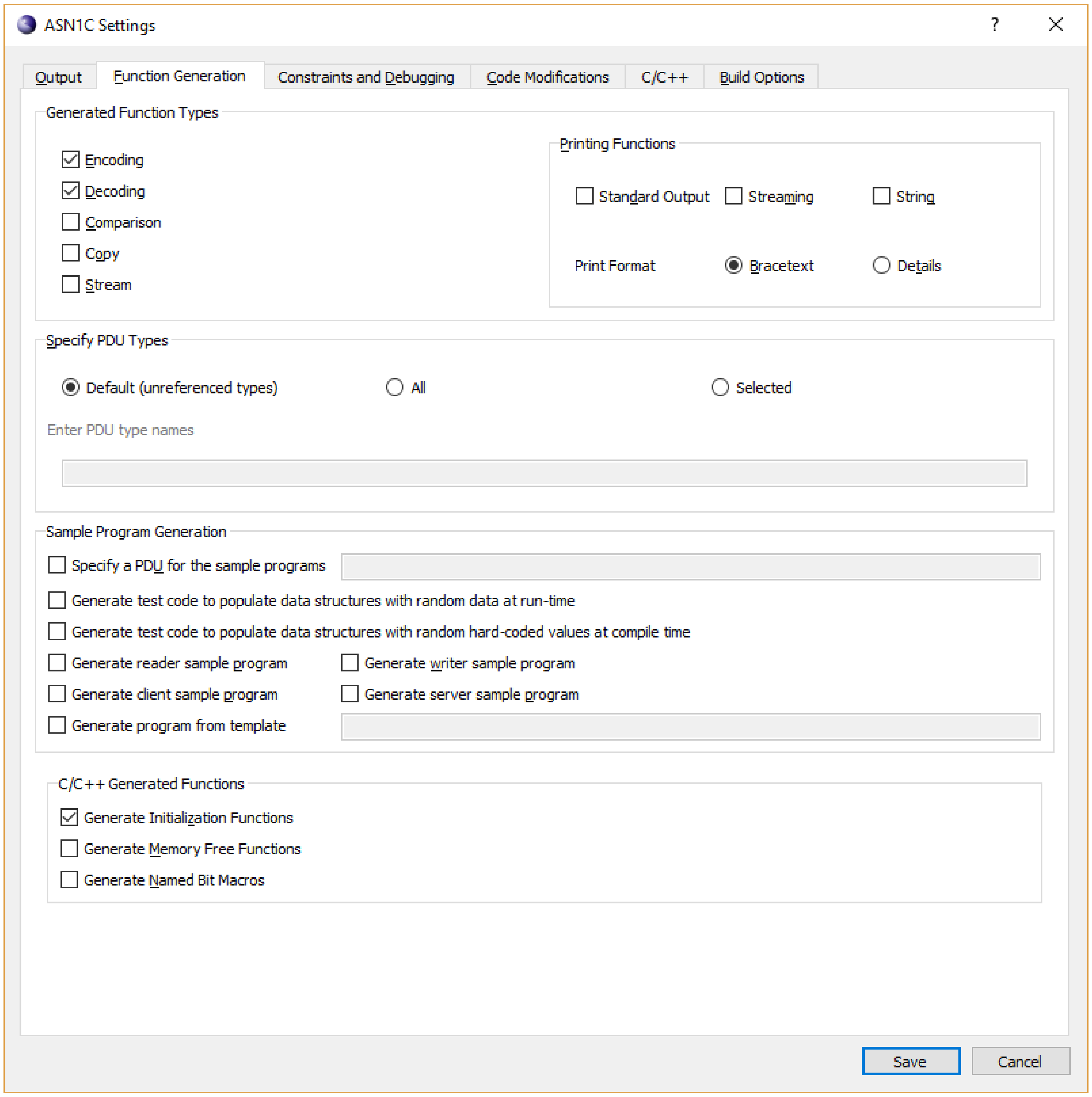

C# Code Organization
☐ Output code to directories based on module names
Code generation option: One .cs file per type
File name:
Binaries Directory
Libraries Directory
Browse
Browse

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



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

Function Generation tab


ASN1C Settings

Output | Function Generation | Constraints and Debugging | Code Modifications | C/C++ | Build Options

Generated Function Types

- [✓] Encoding
- [✓] Decoding
- [ ] Comparison
- [ ] Copy
- [ ] Stream

Printing Functions

- [ ] Standard Output [ ] Streaming [ ] String
- Print Format: [● Bracetext] ○ Details

Specify PDU Types

[● Default (unreferenced types)] ○ All ○ Selected

Enter PDU type names

Sample Program Generation

- [ ] Specify a PDU for the sample programs
- [ ] Generate test code to populate data structures with random data at run-time
- [ ] Generate test code to populate data structures with random hard-coded values at compile time
- [ ] Generate reader sample program [ ] Generate writer sample program
- [ ] Generate client sample program [ ] Generate server sample program
- [ ] Generate program from template

C/C++ Generated Functions

- [✓] Generate Initialization Functions
- [ ] Generate Memory Free Functions
- [ ] Generate Named Bit Macros

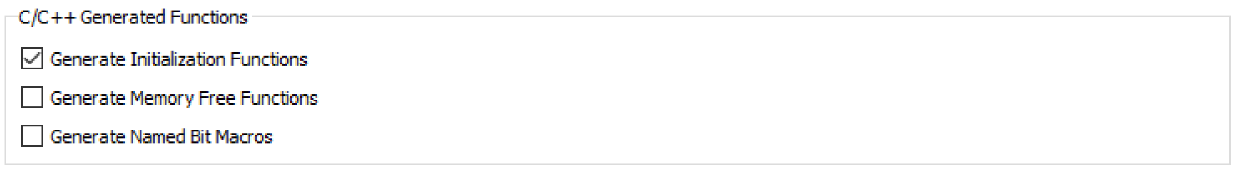
Save Cancel

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

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

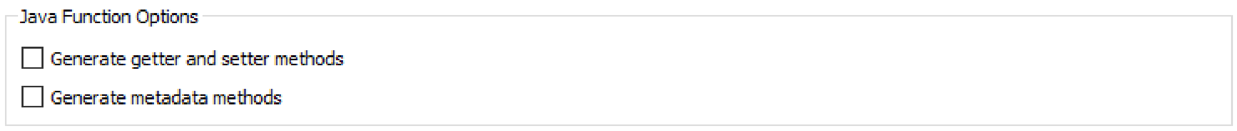
"Sample Program Generation" allows simple encoding and decoding programs, which demonstrate using the generated code, to be generated. The sample writer program can optionally encode randomly-generated test data.

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


C/C++ Generated Functions

- [✓] Generate Initialization Functions
- [ ] Generate Memory Free Functions
- [ ] Generate Named Bit Macros

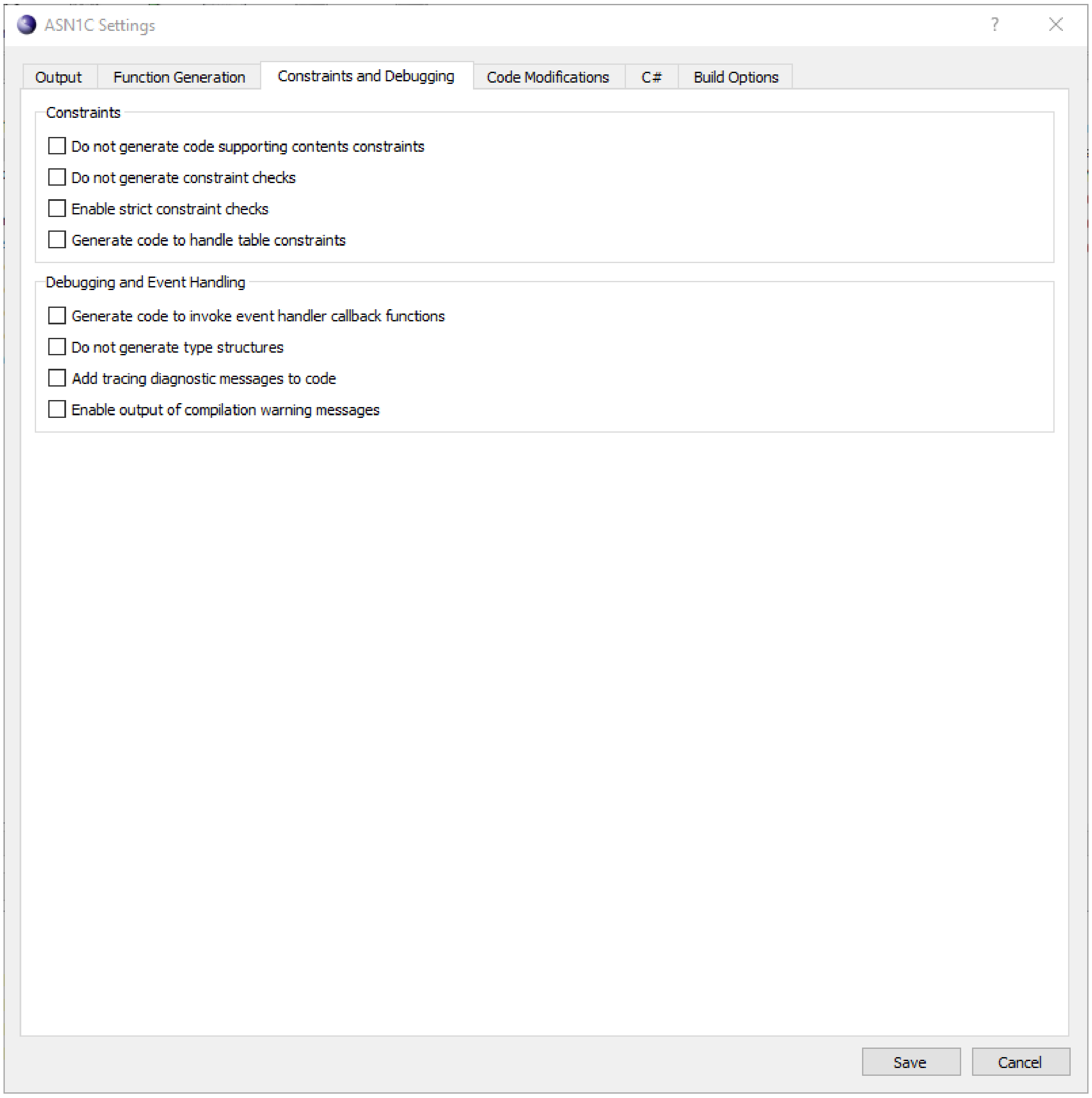
For C and C++, additional functions for memory management and macros for dealing with named bits in BIT STRINGs can be generated. Initialization functions are generated by default. They may be turned off in the window.


Java Function Options

- [ ] Generate getter and setter methods
- [ ] Generate metadata methods

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

Constraints and Debugging tab



Constraints

- [ ] Do not generate code supporting contents constraints
- [ ] Do not generate constraint checks
- [ ] Enable strict constraint checks
- [ ] Generate code to handle table constraints

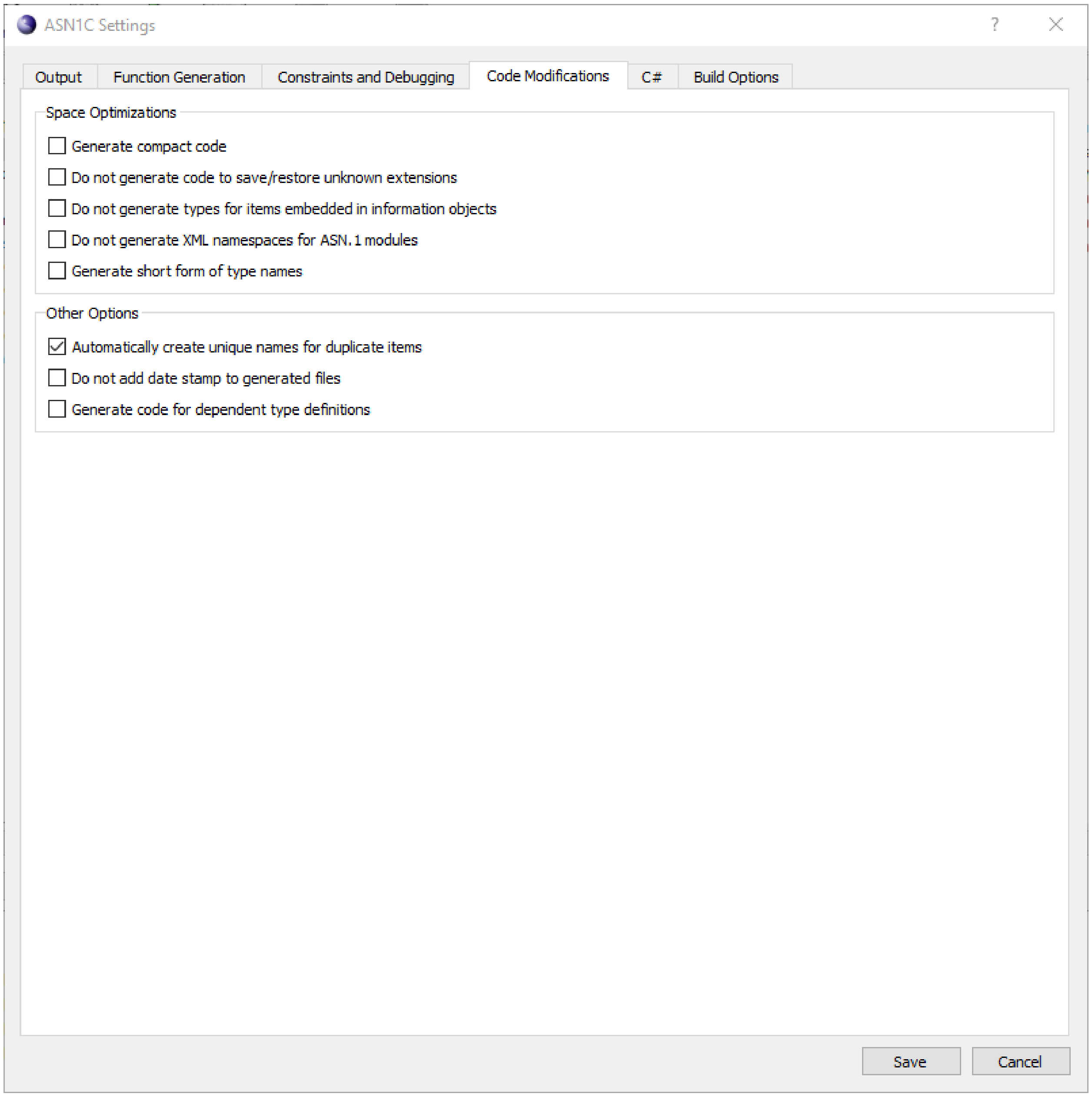
Debugging and Event Handling

- [ ] Generate code to invoke event handler callback functions
- [ ] Do not generate type structures
- [ ] Add tracing diagnostic messages to code
- [ ] Enable output of compilation warning messages

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

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

Code Modifications tab



Space Optimizations

- [ ] Generate compact code
- [ ] Do not generate code to save/restore unknown extensions
- [ ] Do not generate types for items embedded in information objects
- [ ] Do not generate XML namespaces for ASN.1 modules
- [ ] Generate short form of type names

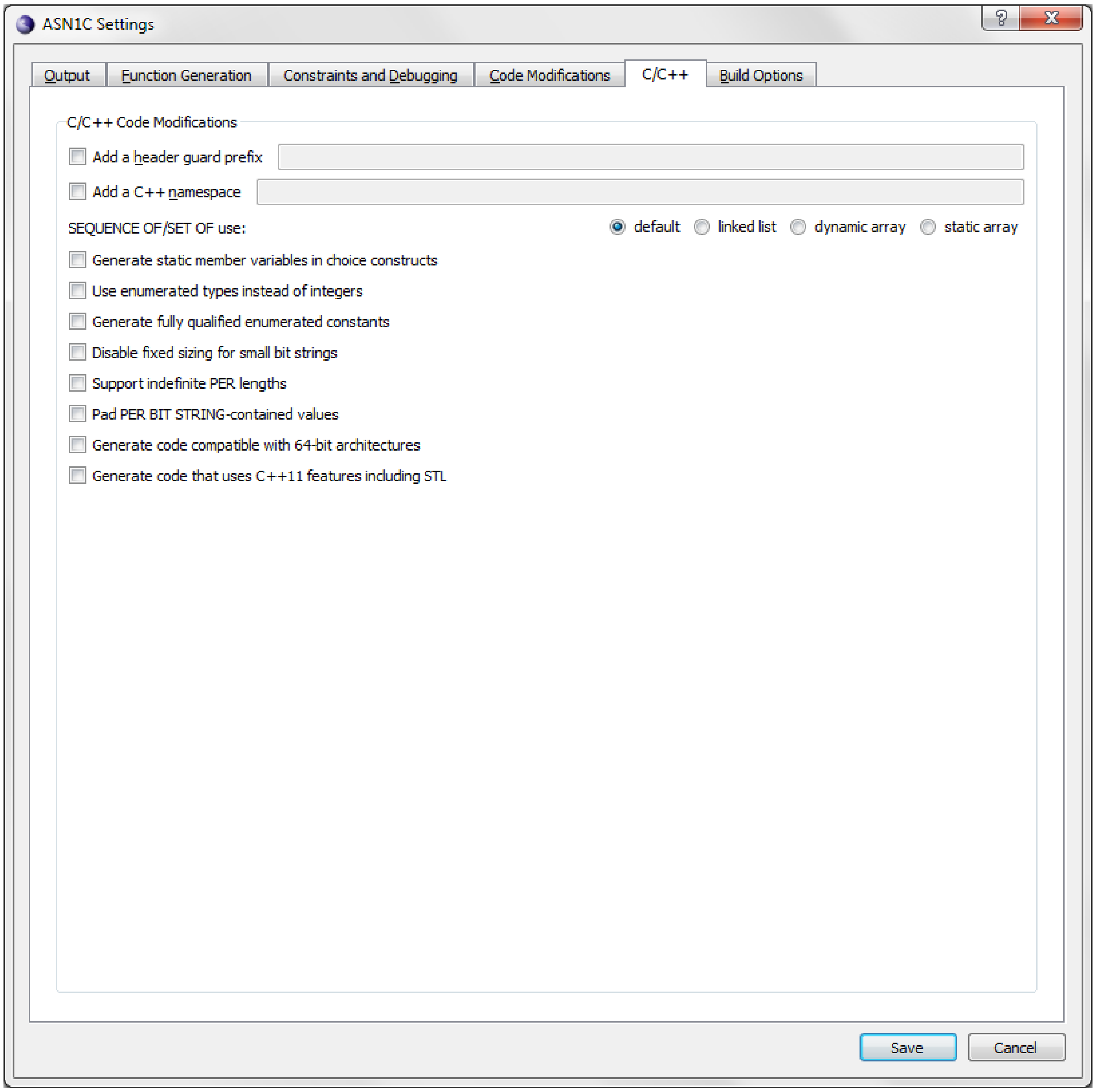
Other Options

- [✓] Automatically create unique names for duplicate items
- [ ] Do not add date stamp to generated files
- [ ] Generate code for dependent type definitions

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

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

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



C/C++ Code Modifications

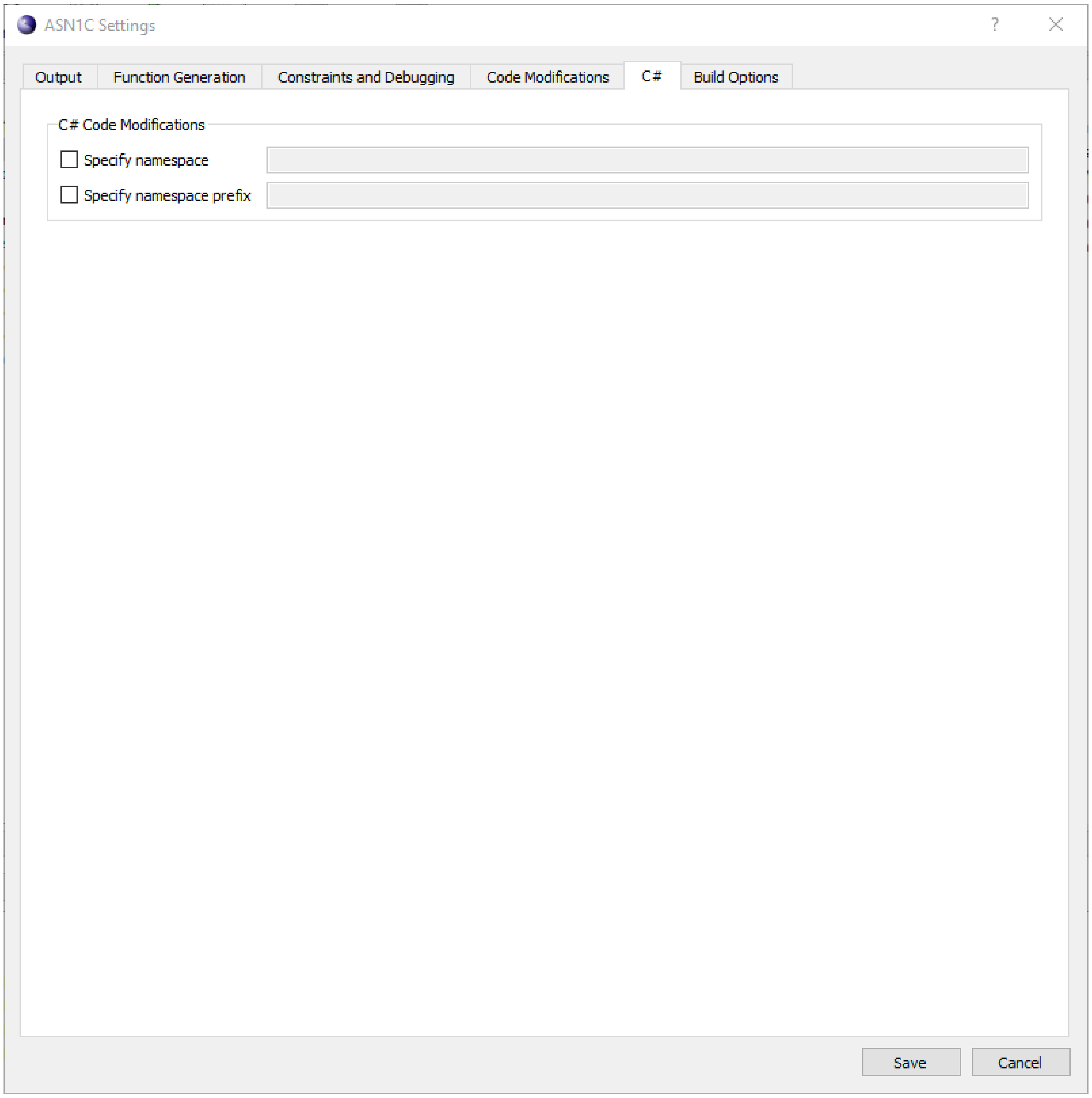
- [ ] Add a header guard prefix
- [ ] Add a C++ namespace

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

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

For C#, the tab is as follows:



C# Code Modifications

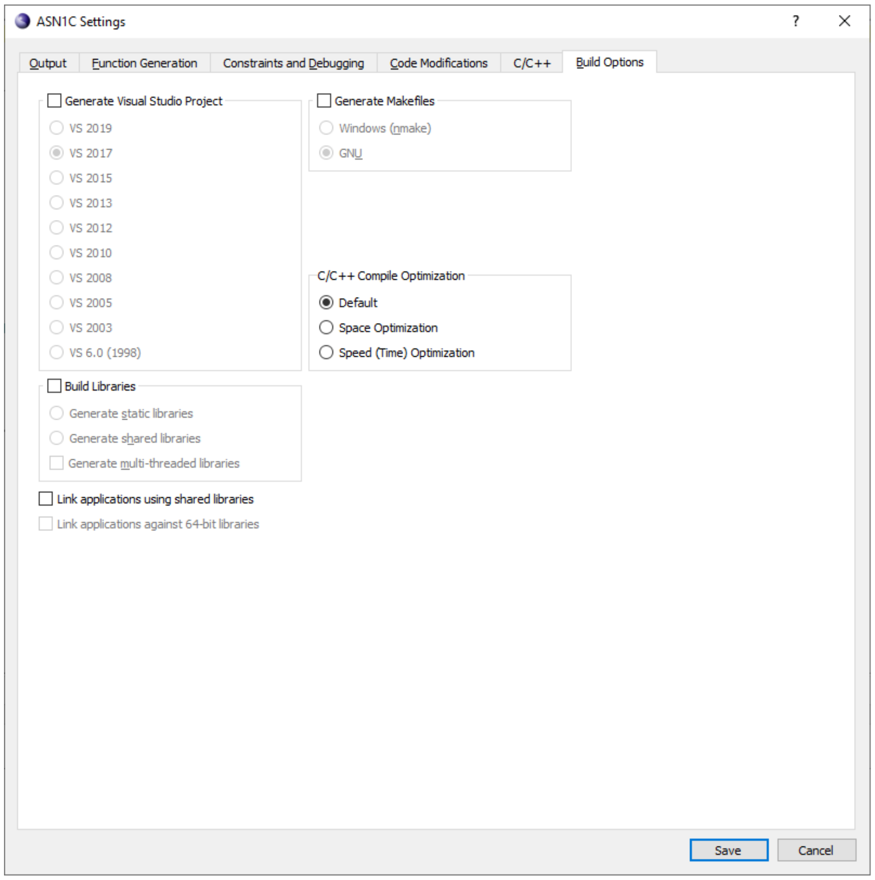
- [ ] Specify namespace
- [ ] Specify namespace prefix

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

Build Options tab

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

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



☐ Generate Visual Studio Project
- ○ VS 2019
- ● VS 2017
- ○ VS 2015
- ○ VS 2013
- ○ VS 2012
- ○ VS 2010
- ○ VS 2008
- ○ VS 2005
- ○ VS 2003
- ○ VS 6.0 (1998)

☐ Generate Makefiles
- ○ Windows (nmake)
- ● GNU

C/C++ Compile Optimization
- ● Default
- ○ Space Optimization
- ○ Speed (Time) Optimization

☐ Build Libraries
- ○ Generate static libraries
- ○ Generate shared libraries
- ☐ Generate multi-threaded libraries

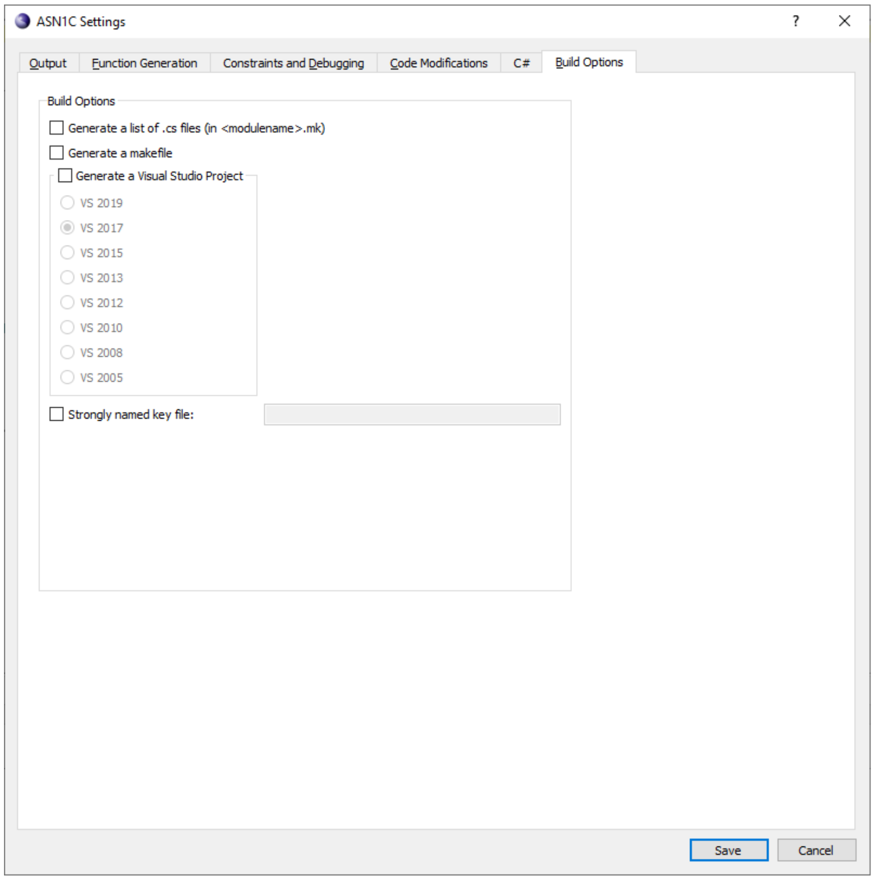
☐ Link applications using shared libraries

☐ 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 "Build Libraries", which will generate the build script to build a library rather than an executable, the desired variety of library can be selected.

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

For C#, the following options are displayed:

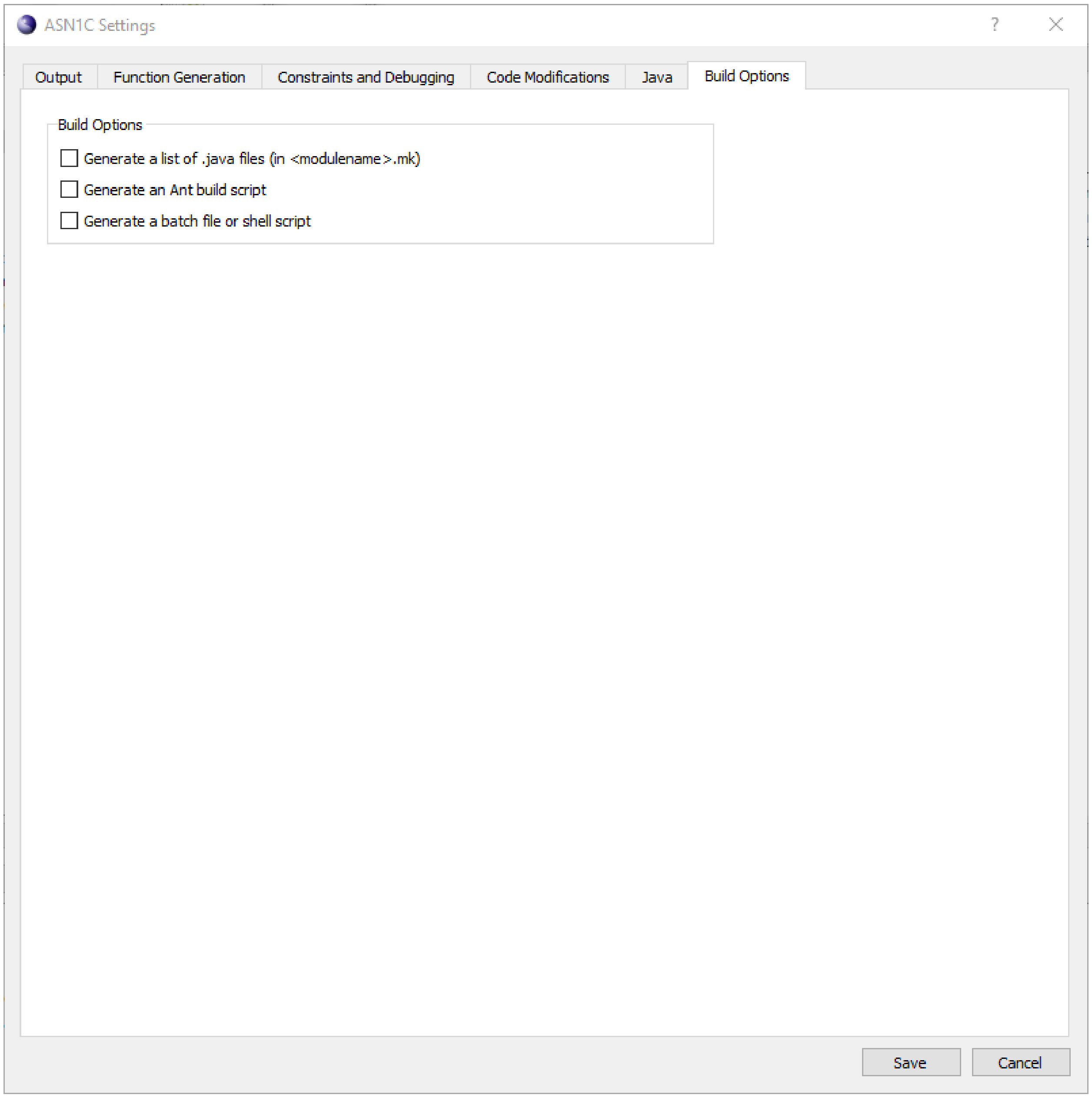


Build Options

- [ ] 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:

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

For Java, the following is displayed:

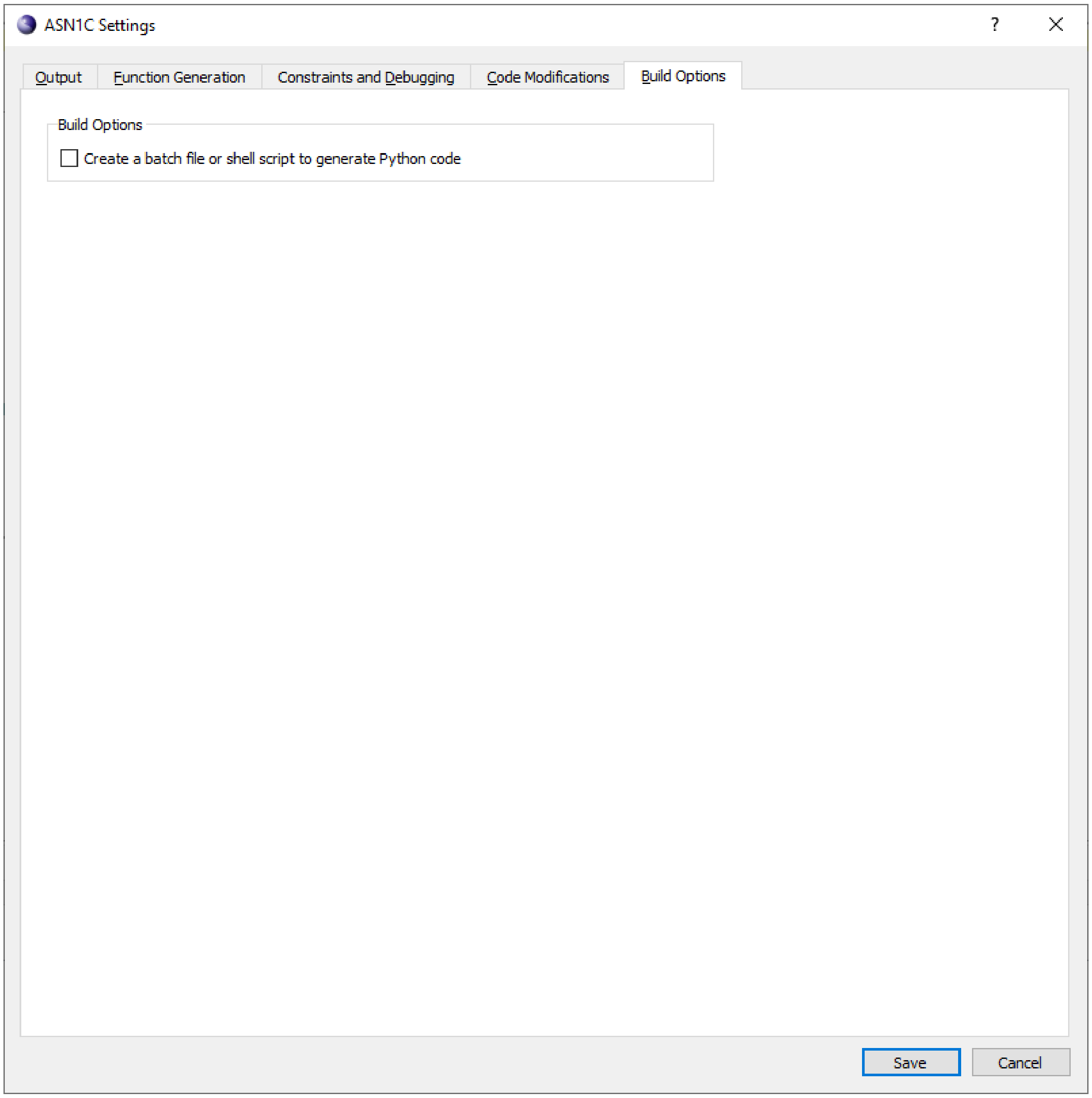


Build Options

- [ ] Generate a list of .java files (in <modulename>.mk)
- [ ] Generate an Ant build script
- [ ] Generate a batch file or shell script

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

For Python, the following is displayed:

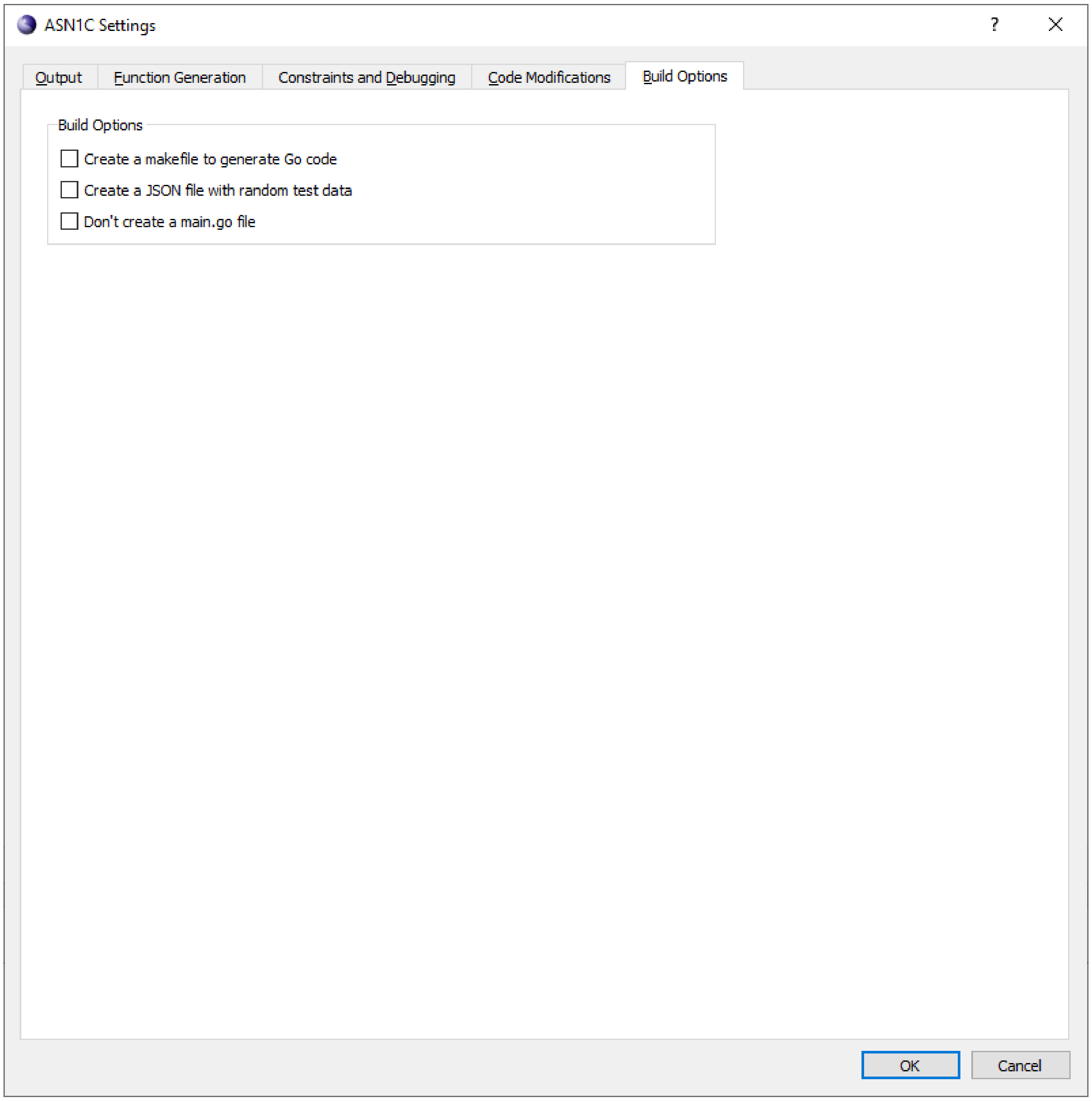


Build Options

☐ Create a batch file or shell script to generate Python code

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

For Go, the following is displayed:



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. Telling the generator not to create a main.go file (for instance, if there already is one that has been modified) is also an option.

Chapter 4. Generated Go Source Code

A Go source file with extension '.go' will be generated for most ASN.1 modules defined in the ASN.1 source file. If a module does not contain any ASN.1 types that would trigger generating a Go type or value, then no Go file is generated.

General Hierarchy of Generated Go Source Files

When Go code is generated, a standard directory structure is created to hold the generated files which form a Go module. At the root level, the main.go and go.mod files are generated. Under that is asn1gen subdirectory, in which the generated Go files are stored. Below the asn1gen subdirectory is an asn1rt subdirectory, in which the required base run-time source files are stored. Executing “go build” will build the full package.

General Form of a Generated Go Source File

A generated Go file begins with `package asn1gen`. All generated Go code is included in the asn1gen package.

After that is an import statement, which will import the base run-time package, asn1rt, and any required built-in Go packages. This is followed by type and constant definitions. See the “ASN.1 To Go Type Mappings” section for details on the mapping of ASN.1 types to Go types.

For PER, type-specific encode and decode functions are generated. For JSON, custom marshal and unmarshal functions are generated for some types. The built-in Go marshaling logic is used wherever possible to do the JSON conversion.

Chapter 5. Generated Go Main File (main.go)

Go code is normally generated from within the golang subdirectory in an ASN1C installation. At the main level, main.go and go.mod files are generated. The main.go file is a template file for using the files in the asn1gen package. The asn1gen package source is generated into an asn1gen subdirectory beneath the top-level directory. The main.go file contains a sample writer section to encode a data record of the chosen Protocol Data Unit (PDU) type and to a file. The reader section reads decodes from the file and prints the results. Each is invoked by running the main program with an argument of 'writer' or 'reader', respectively.

By default, the main.go writer section contains a TODO comment to remind the user that a data variable of the PDU type needs to be populated prior to encoding. If the *-test* command-line option is specified, code will be added to populate the PDU variable with random test data.

The procedure to generate and build Go source code is rather straightforward:

1. Invoke the asn1c compiler on your ASN.1 source file(s) to generate Go source code. For example: asn1c test.asn -go -per

2. Invoke 'go build' to run the Go compiler to build the generated package.

That is it. If successful, an executable file should have been created which can be run with the writer or reader option.

Note that it is possible to use the *-make* command-line option to generate a makefile for future compilations. The user should also be careful to use the *-no-go-main* option on future compilations so as to not regenerate the main.go file in the event it was edited by the user.

Chapter 6. ASN.1 To Go Type Mappings

Type Mappings

BOOLEAN

A Go type is generated for the ASN.1 BOOLEAN type which equates the ASN.1 type assignment to the built-in Go *bool* type. For example:

ASN.1:

```
MyBool ::= BOOLEAN
```

Generated Go code:

```
type MyBool bool
```

If a BOOLEAN type is referenced directly in an element within a constructed type, the Go bool type is referenced directly.

INTEGER

A Go type is generated for the ASN.1 INTEGER type which equates the ASN.1 type assignment to a built-in Go 64-bit integer type. For an unconstrained integer type, the Go type used is a signed 64-bit integer (int64). For example:

ASN.1:

```
MyInt ::= INTEGER
```

Generated Go code:

```
type MyInt int64
```

If a constraint constrains the value to a nonnegative range, an unsigned 64-bit integer type is used (uint64):

ASN.1:

```
MyUInt ::= INTEGER (0..9)
```

Generated Go code:

```
type MyInt uint64
```

Note that no attempt is made to used smaller integer types for smaller ranges.

If an INTEGER type is referenced directly in an element within a constructed type, the Go integer type is referenced directly.

If the INTEGER type contains a list of named numbers, Go constants are generated for each number, the same as is done for the ASN.1 ENUMERATED type.

BIT STRING

The ASN.1 BIT STRING type is mapped to the Go BitString type defined within the ASN.1 run-time (asn1rt.BitString). This is in turn mapped to the encoding/asn1.BitString from the Go standard library. For example:

ASN.1:

```
MyBitStr ::= BIT STRING
```

Generated Go code:

```
type MyBitStr asn1rt.BitString
```

Bits are specified in the Bytes field in most-significant bit order. For example, the bit string '1'B (consisting of a single bit which is set) would be represented using BitLength = 1 and Bytes being a slice of 1 byte, set to 0x80.

Named Bits

In the ASN.1 standard, it is possible to define a named bit string that specifies names for different bit positions. ASN1C provides support for this type by generating Go constants and run-time functions that can be used to set, clear, or test these named bits. These constants equate the bit name to the bit number defined in the specification. They can be used with the *SetBit*, *ClearBit*, and *TestBit* run-time functions to set, clear, and test the named bits.

Contents Constraint

It is possible to specify a contents constraint on a BIT STRING type using the CONTAINING keyword. This indicates that the encoded contents of the specified type should be packed within the BIT STRING container. An example of this type of constraint is as follows:

```
ContainingBS ::= BIT STRING (CONTAINING INTEGER)
```

ASN1C will generate a type definition that references the type that is within the containing constraint. In this case, that would be INTEGER; therefore, the Go type for INTEGER would be used, which is int64. This direct use of the containing type can be suppressed through the use of the *-noContaining* command-line argument. In that case, a normal BIT STRING type will be used and it will be the user's responsibility to do the necessary packing and unpacking operations to encode and decode the variable correctly.

OCTET STRING

The ASN.1 OCTET STRING type is mapped to the Go OctetString type defined within the ASN.1 run-time (asn1rt.OctetString). This is in turn mapped to a byte slice. For example:

ASN.1:

```
MyOctStr ::= OCTET STRING
```

Generated Go code:

```
type MyOctStr asn1rt.OctetString
```

Contents Constraint

It is possible to specify a contents constraint on an OCTET STRING type using the CONTAINING keyword. This indicates that the encoded contents of the specified type should be packed within the OCTET STRING container. An example of this type of constraint is as follows:

```
ContainingOS ::= OCTET STRING (CONTAINING INTEGER)
```

ASN1C will generate a type definition that references the type that is within the containing constraint. In this case, that would be INTEGER; therefore, the Go type for INTEGER would be used, which is int64. This direct use of the containing type can be suppressed through the use of the *-noContaining* command-line argument. In that case, a normal OCTET STRING type will be used and it will be the user's responsibility to do the necessary packing and unpacking operations to encode and decode the variable correctly.

ENUMERATED

The Go mapping for an ASN.1 ENUMERATED type is the same as for INTEGER. By default, uint64 is used. If the ENUMERATED type contains negative-value identifiers, then int64 is used. Go constants are generated for each of the enumerated identifiers. The constant names consist of the type name concatanated with the enumerated identifier name in order to provide unique names. Note that for Go, camel-case is used in the generated identifiers and underscores are not used.

If the enumerated type is extensible, a special UNKNOWN identifier constant is added to the enumerated contant list.

If an enumerated type is referenced directly in an element within a constructed type, a compiler-generated type is generated for it consisting of the constructed type name concatanated with the element name to form a camel-cased identifer. This identifier is then used as the prefix for generated enumerated identifiers.

Example - Enumerated Type Assignment

ASN.1:

```
   MyEnum ::= ENUMERATED { red(0), green(1), blue(2), ... }
```

Generated Go code:

```
   type MyEnum uint64
const (
 MyEnumRed = 0
 MyEnumGreen = 1
 MyEnumBlue = 2
 MyEnumUNKNOWN = 3
)
```

Example - Enumerated Type Referenced in Element

ASN.1:

```
   MySeq ::= SEQUENCE {
     enum ENUMERATED { a, b, c }
   }
```

Generated Go code:

```
   type MySeqEnum uint64
   const (
      MySeqEnumA = 0
      MySeqEnumB = 1
      MySeqEnumC = 2
   )

   type MySeq struct {
      Enum MySeqEnum
   }
```

NULL

A Go type is generated for the ASN.1 NULL type which equates the ASN.1 type assignment to the built-in Go *bool* type. The purpose of having a type for it is so that it can act as a placeholder to denote the presence or absense of this type within a constructed type. For example, if it used in an optional element or within a choice construct.

If a NULL type is referenced directly in an element within a constructed type, the Go bool type is referenced directly.

Example

ASN.1:

```
MyNull ::= NULL
```

Generated Go code:

```
type MyNull bool
```

OBJECT IDENTIFIER

The ASN.1 OBJECT IDENTIFIER type is mapped to the Go ObjectIdentifier type defined within the ASN.1 run-time (asn1rt.ObjectIdentifier). This in turn is mapped to a uint64 slice (*[]uint64*), with each uint64 representing a single subidentifier. For example:

ASN.1:

```
MyOID ::= OBJECT IDENTIFIER
```

Generated Go code:

```
type MyOID asn1rt.ObjectIdentifier
```

RELATIVE-OID

The ASN.1 RELATIVE-OID type mapping is the same as described above for OBJECT IDENTIFIER.

REAL

The ASN.1 REAL type is mapped to the Go Real type defined within the ASN.1 run-time (asn1rt.Real). This in turn is mapped to is mapped to the Go built-in float64 type.

If the REAL type is constrained to base 10 values, it is mapped to the Go RealBase10 type defined within the ASN.1 run-time (asn1rt.RealBase10). This in turn is mapped to is mapped to the Go built-in string type.

Example

ASN.1:

```
MyReal ::= REAL
MyReal10 ::= REAL(0 | WITH COMPONENTS { ..., base (10) })
```

Generated Go code:

```
type MyReal asn1rt.Real
type MyReal10 asn1rt.RealBase10
```

SEQUENCE/SET

An ASN.1 SEQUENCE or SET type is a constructed type consisting of a series of element definitions; each element has an assigned ASN.1 type. An ASN.1 SEQUENCE or SET is mapped to a Go struct type with each ASN.1 element having an equivalent Go element definition. The general form is as follows:

ASN.1 production:

```
<name> ::= SEQUENCE {
<element1-name> <element1-type>,
<element2-name> <element2-type>,
...
}
```

Generated Go code:

```
type <name> struct {
<element1-name> <type1>
<element2-name> <type2>
...
}
```

The `<type1>` and `<type2` placeholders represent the Go types for the ASN.1 types `<element1-type>` and `<element1-type>`, respectively.

Here is an example:

ASN.1:

```
A ::= SEQUENCE {
  x  SEQUENCE {
     a1 INTEGER,
     a2 BOOLEAN
  },
  y OCTET STRING (SIZE (10))
  }
```

Generated Go code:

```
type AX struct {
   A1 int64
   A2 bool
}

type A struct {
   X AX
   Y asn1rt.OctetString
}
```

First, note that the Go field names in the struct begin with uppercase letters. This means the names are exported for use outside of the package in which they are defined.

Second, notice that we generated type AX, even though no such type was defined in the ASN.1. This is the result of using a nested, constructed type in the ASN.1. The elements in a SEQUENCE or SET can be of any ASN.1 type, including other constructed types. Sometimes, one constructed type definition is nested inside another, rather than defining a separate type and referencing it by name. For example, it is possible to nest a SEQUENCE definition within another SEQUENCE definition (see element x in the above example).

The ASN1C compiler first recursively pulls all nested constructed elements out of the SEQUENCE and forms new internal types. The internal ASN.1 names of these types are of the form `<name>_<element-name1>_<elementname2>_ ...<element-nameN>`. In the above example, the compiler internally created a type for element x: `A_x`. The compiler then generates code as if the ASN.1 had been written without nesting, using the type definitions it created internally. Transforming the ASN.1 name to a Go name produced `AX`.

In the case of nesting levels greater than two, all of the intermediate element names are used to form the final name. For example, consider the following type definition that contains three nesting levels:

```
X ::= SEQUENCE {
   a SEQUENCE {
      aa SEQUENCE { x INTEGER, y BOOLEAN },
      bb INTEGER
   }
}
```

In this case, the generation of compiler-generated types results in the following equivalent type definitions:

```
X-a-aa ::= SEQUENCE { x INTEGER, y BOOLEAN }

X-a ::= SEQUENCE { aa X-a-aa, bb INTEGER }

X ::= SEQUENCE { X-a a }
```

Note that the name for the `aa` element type is `X-a-aa`. It contains both the name for `a` (at level 1) and `aa` (at level 2). The concatanation of all of the intermediate element names can lead to very long names in some cases. To get around the problem, the *-shortnames* command-line option can be used to form shorter names. In this case, only the type name and the last element name are used. In the example above, this would lead to an element name of `X-aa`. The disadvantage of this is that the names may not always be unique. If using this option results in non-unique names, an `_n` suffix is added where `n` is a sequential number to make the names unique.

Although the compiler can handle embedded constructed types within productions, it is generally not considered good style to define productions this way. It is much better to manually define the constructed types for use in the final production definition. For example, the production defined at the start of this section can be rewritten as the following set of productions:

```
X ::= SEQUENCE {
   a1 INTEGER,
   a2 BOOLEAN
}

A ::= SEQUENCE {
   x X,
   y OCTET STRING (SIZE (10))
}
```

This makes the generated code easier to understand for the end user.

OPTIONAL keyword

Elements within a sequence can be declared to be optional using the OPTIONAL keyword. This indicates that the element is not required. Optional elements are handled in Go using pointer types. If the pointer to the element value is nil, it means the element is not present.

For example, the elements in the following production are marked as optional:

```
Aseq ::= SEQUENCE {
   x      INTEGER OPTIONAL,
   y      BOOLEAN OPTIONAL
}
```

In this case, the following Go type is generated:

```
type Aseq struct {
   X *int64
   Y *bool
}
```

When this structure is populated for encoding, if the element is present, the developer must set the pointer value to point at a value. This can be done using `new` or by assigning the value to a local variable and then storing the address of the variable in the generated structure. Conversely, when a message is decoded into this structure, the developer must test to see if the elements are nil to see if they were present in the message.

DEFAULT keyword

The DEFAULT keyword allows a default value to be specified for elements within the SEQUENCE. ASN1C will generate a non-pointer type for elements using DEFAULT if the type is one of several simple types (BOOLEAN, INTEGER, ENUMERATED, REAL, BIT STRING, OCTET STRING, and character string types). In other cases, ASN1C will generate a pointer type, the same as it does for an element with OPTIONAL; a nil pointer is interpreted as representing the default value.

For PER, elements set to their default value MUST be encoded as ABSENT. In the cases where ASN1C generates a non-pointer type, the compiler automatically generates code to meet this requirement. In the cases where ASN1C generates a pointer type, the programmer must set the pointer to nil rather than populate the field with the default value.

Extension Elements

If the SEQUENCE type contains an open extension field (i.e., a ... at the end of the specification or a ..., ... in the middle), a special element will be inserted to capture encoded extension elements for inclusion in the final encoded message. This element will be of type *[][]byte* and have the name *ExtElem1*. This holds an array of encoded components.

The *-noOpenExt* command line option can be used to alter this default behavior. If this option is specified, the *ExtElem1* element is not included in the generated structure; any extension data that may be present in a decoded message is skipped.

If the SEQUENCE type contains an extension marker and extension elements, then the actual extension elements will be present in addition to the *ExtElem1* element. These elements will be treated as optional elements whether they were declared that way or not. The reason is because a version 1 message could be received that does not contain the elements.

If version brackets are present, a struct is generated for each extension element group. This struct has type name of the form *<container>ExtGrpV<n>* where <container> is the name of the container type and <n> is the version number of the group. This type is then referenced as an optional element in the container type.

An example of a type with version brackets is as follows:

```
TestSequence ::= SEQUENCE {
   item-code    INTEGER (0..254),
   item-name    IA5String (SIZE (3..10)) OPTIONAL,
   ... ! 1,
   urgency      ENUMERATED { normal, high } DEFAULT normal,
   [[ alternate-item-code       INTEGER (0..254),
      alternate-item-name       IA5String (SIZE (3..10)) OPTIONAL
   ]]
}
```

In this case, a structure named `TestSequenceExtGrpV3` is generated for the items in the version brackets. The number 3 is used because 2 would be the version extension number of the urgency field. The generated Go types for this construct would be as follows:

```
   type TestSequenceExtGrpV3 struct {
 AlternateItemCode uint64
 AlternateItemName *string
   }

   type TestSequence struct {
 ItemCode uint64
 ItemName *string
 Urgency *uint64
 ExtGrpV3 *TestSequenceExtGrpV3
 ExtElem1 [][]byte
   }
```

The presence or absence of the entire version block would be determined by whether the ExtGrpV3 element pointer is set.

SEQUENCE OF/SET OF

An ASN.1 SEQUENCE OF or SET OF type is mapped to a Go slice. The general form is *[]<element type>* where <element type> would be replaced with the Go type of the SEQUENCE OF element.

SEQUENCE or SET OF Element in a Constructed Type

If an element in a SEQUENCE, SET, or CHOICE directly references a SEQUENCE OF or SET OF type, a slice type is generated inline. A separate type is not generated to be referenced by the element. For example, consider the following type definitions:

```
   SeqOfInt ::= SEQUENCE OF INTEGER

   Seq ::= SEQUENCE {
      a SeqOfInt,
      b SEQUENCE OF BOOLEAN
   }
```

This results in the following Go types being generated:

```
   type SeqOfInt []int64

   type Seq struct {
 A SeqOfInt
 B []bool
   }
```

Note that a separate type is not generated for the inline SEQUENCE OF BOOLEAN, the bool slice is generated inline.

SEQUENCE OF Constructed Type

As with other constructed types, the `<Type>` in `SEQUENCE OF Type` can be any ASN.1 type, including constructed types. Therefore, it is possible to have a SEQUENCE OF SEQUENCE, SEQUENCE OF CHOICE, etc.

In such cases, the compiler interally creates an ASN.1 type for the nested, constructed type. The format of the name for this type is as follows:

```
<ProdName>Element
```

In this definition, `<ProdName>` refers to the Go name of the production containing the SEQUENCE OF type.

For example, a simple (and very common) single level nested SEQUENCE OF construct might be as follows:

```
A ::= SEQUENCE OF SEQUENCE { a INTEGER, b BOOLEAN }
```

In this case, a type is generated for the element of the SEQUENCE OF production. Internally, this is transformed into the equivalent ASN.1:

```
A-element ::= SEQUENCE { a INTEGER, b BOOLEAN }

A ::= SEQUENCE OF A-element
```

These types are then converted into the equivalent Go types using the standard mapping that was previously described.

SET OF

The ASN.1 SET OF type is mapped to a slice type, just the same as described previously for SEQUENCE OF.

CHOICE

The ASN.1 CHOICE type is converted into a Go struct containing an integer for the choice tag value (`T`) followed by a struct (`U`) with a field for each of the CHOICE elements. This differs from C/C++, where a union type is used for the set of alternatives. There is no union type in Go, so a struct containing pointer values for all of the alternatives is used. The `T` field indicates which of the alternatives is chosen and which of the fields in the `U` struct shall be non-nil; exactly one field shall be non-nil at a time.

The tag value is simply a sequential number starting at one for each alternative in the CHOICE. A Go constant is generated for each of these values. The format of this constant is `<name><element-name>TAG` where `<name>` is the name of the ASN.1 production and `<element-name>` is the name of the CHOICE alternative.

The fields of the `U` struct, corresponding to the alternatives for the CHOICE type, are generated much the same as for the elements of a SEQUENCE type (described earlier), except that pointer types are used. The compiler internally pulls out any nested constructed types and creates an ASN.1 type for it, and that is reflected in the generated code.

ASN.1 production:

```
<name> ::= CHOICE {
   <element1-name> <element1-type>,
   <element2-name> <element2-type>,
   ...
}
```

Generated Go code:

```
const (
    <name>_<element1-name>TAG = 1
    <name>_<element2-name>TAG = 2
    ...
)

type <name> struct {
    T uint64
    U struct {
        <type1> *<element1-name>
        <type2> *<element2-name>
        ...
    }
}
```

Open Type

An *Open Type* as defined in the X.680 standard is specified as a reference to a *Type Field* in an *Information Object Class*. The most common form of this is when the *Type* field in the built-in TYPE-IDENTIFIER class is referenced as follows:

```
TYPE-IDENTIFIER.&Type
```

See the section in this document on Information Objects for a more detailed explanation.

The *Open Type* is converted into a Go byte slice ([]byte). The slice is assumed to contain previously encoded ASN.1 data. When a message containing an open type is decoded, a copy of the encoded data is made.

If the *-tables* or *-table-unions* command line option is selected and the ASN.1 type definition references a table constraint, the code generated is different. In this case, instead of a byte slice, a union-type structure is used, similar to what is done for a CHOICE type. Each information object in the information object set referenced by the table constraint specifies an actual ASN.1 type that can be used for the open type; taken together, these objects represent all of the possible alternatives for the open type.

Character String Types

All ASN.1 character string types, including multi-byte types such as BMPString and UniversalString, are mapped to the Go string type. The Go string will be interpreted by the generated code as UTF-8.

The useful character string types in ASN.1 are as follows:

```
UTF8String        ::=  [UNIVERSAL 12]  IMPLICIT OCTET STRING
NumericString     ::=  [UNIVERSAL 18]  IMPLICIT IA5String
PrintableString   ::=  [UNIVERSAL 19]  IMPLICIT IA5String
T61String         ::=  [UNIVERSAL 20]  IMPLICIT OCTET STRING
VideotexString    ::=  [UNIVERSAL 21]  IMPLICIT OCTET STRING
IA5String         ::=  [UNIVERSAL 22]  IMPLICIT OCTET STRING
UTCTime           ::=  [UNIVERSAL 23]  IMPLICIT GeneralizedTime
GeneralizedTime   ::=  [UNIVERSAL 24]  IMPLICIT IA5String
GraphicString     ::=  [UNIVERSAL 25]  IMPLICIT OCTET STRING
VisibleString     ::=  [UNIVERSAL 26]  IMPLICIT OCTET STRING
GeneralString     ::=  [UNIVERSAL 27]  IMPLICIT OCTET STRING
UniversalString   ::=  [UNIVERSAL 28]  IMPLICIT OCTET STRING
BMPString         ::=  [UNIVERSAL 30]  IMPLICIT OCTET STRING
```

```
ObjectDescriptor  ::=  [UNIVERSAL 7]  IMPLICIT GraphicString
```

Time String Types

The ASN.1 *GeneralizedTime* and *UTCTime* types are mapped to the Go string type.

EXTERNAL

The ASN.1 EXTERNAL type is a useful type used to include non-ASN.1 or other data within an ASN.1 encoded message. This type is described using the following ASN.1 SEQUENCE:

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

The ASN.1 EXTERNAL type is mapped to a Go struct, generated from the above SEQUENCE definition.

EMBEDDED PDV

The ASN.1 EMBEDDED PDV type is a useful type used to include non-ASN.1 or other data within an ASN.1 encoded message. This type is described using the following ASN.1 SEQUENCE:

```
EmbeddedPDV ::= [UNIVERSAL 11] IMPLICIT SEQUENCE {
   identification CHOICE {
      syntaxes SEQUENCE {
         abstract OBJECT IDENTIFIER,
         transfer OBJECT IDENTIFIER
      },
      syntax OBJECT IDENTIFIER,
      presentation-context-id INTEGER,
      context-negotiation SEQUENCE {
         presentation-context-id INTEGER,
         transfer-syntax OBJECT IDENTIFIER
      },
      transfer-syntax OBJECT IDENTIFIER,
      fixed NULL
   },
   data-value OCTET STRING
}
```

The ASN.1 EMBEDDED PDV type is mapped to a Go struct, generated from the above SEQUENCE definition.

Parameterized Types

The ASN1C compiler can parse parameterized type definitions and references as specified in the X.683 standard. These types allow dummy parameters to be declared that will be replaced with actual parameters when the type is referenced. This is similar to templates in C++.

A simple and common example of the use of parameterized types is the declaration of an upper bound on a sized type as follows:

```
SizedOctetString{INTEGER:ub} ::= OCTET STRING (SIZE (1..ub))
```

In this definition, `ub` would be replaced with an actual value when the type is referenced. For example, a sized octet string with an upper bound of 32 would be declared as follows:

```
OctetString32 ::= SizedOctetString{32}
```

The compiler would handle this just as if the original type was declared to be an octet string of size 1 to 32. That is, it will map the type to a go byte slice ([]byte) and do constraint checks before encoding and after decoding to ensure the value is within the specified range.

Another common example of parameterization is the substitution of a given type inside a common container type. For example, security specifications frequently contain a 'signed' parameterized type that allows a digital signature to be applied to other types. An example of this is as follows:

```
SIGNED { ToBeSigned } ::= SEQUENCE {
   toBeSigned    ToBeSigned,
   algorithmOID  OBJECT IDENTIFIER,
   paramS        Params,
   signature     BIT STRING
}
```

An example of a reference to this definition would be as follows:

```
SignedName ::= SIGNED { Name }
```

where `Name` would be another type defined elsewhere within the module.

The compiler performs the substitution to create the proper Go type for `SignedName`:

```
type SignedName struct {
    ToBeSigned Name
    AlgorithmID []uint64
    ParamS Params
    Signature asn1.BitString
}
```

When processing parameterized type definitions, the compiler will first look to see if the parameters are actually used in the final generated code. If not, they will simply be discarded and the parameterized type converted to a normal type reference. For example, when used with information objects, parameterized types are frequently used to pass information object set definitions to impose table constraints on the final type. Since table constraints do not affect the code that is generated by the compiler when table constraint code generation is not enabled, the parameterized type definition is reduced to a normal type definition and references to it are handled in the same way as defined type references. This can lead to a significant reduction in generated code in cases where a parameterized type is referenced over and over again.

For example, consider the following oft-repeated pattern from the UMTS 3GPP specifications:

```
ProtocolIE-Field {RANAP-PROTOCOL-IES : IEsSetParam} ::= SEQUENCE {
   id              RANAP-PROTOCOL-IES.&id          ({IEsSetParam}),
   criticality     RANAP-PROTOCOL-IES.&criticality ({IEsSetParam}{@id}),
   value           RANAP-PROTOCOL-IES.&Value       ({IEsSetParam}{@id})
}
```

In this case, `IEsSetParam` refers to an information object set specification that constrains the values that are allowed to be passed for any given instance of a type referencing a `ProtocolIE-Field`. The compiler does not add any extra code to check for these values, so the parameter can be discarded (note that this is not true if the *-tables* compiler option is specified). After processing the Information Object Class references within the construct (refer to the section on *Information Objects* for information on how this is done), the reduced definition for `ProtocolIE-Field` becomes the following:

```
ProtocolIE-Field ::= SEQUENCE {
   id ProtocolIE-ID,
   criticality Criticality,
   value ASN.1 OPEN TYPE
}
```

References to the field are simply replaced with a reference to the `ProtocolID-Field` type.

If *-tables* is specified, the parameters are used and a new type instance is created in accordance with the rules above.

Value Mappings

ASN1C can parse any type of ASN.1 value specification, but it will only generate code for following value specifications:

- BOOLEAN
- INTEGER
- REAL
- ENUMERATED
- Binary String
- Hexadecimal String
- Character String
- OBJECT IDENTIFER

ASN.1 value specifications are mapped to Go constant declarations of various types with the name transformed to start with prefix 'Asn1v' followed by the value name transformed as per Go naming rules.

BOOLEAN Value

A BOOLEAN value is mapped to a Go true or false constant declaration:

ASN.1 production:

```
<name> BOOLEAN ::= <value>
```

Generated code:

```
const Asn1v<name> bool = <value>;
```

ASN.1 BOOLEAN values are the uppercase names TRUE and FALSE. The equivalent Go names are true and false.

INTEGER Value

An INTEGER type is mapped to a Go integer value constant:

ASN.1 production:

```
<name> INTEGER ::= <value>
```

Generated code:

```
const Asn1v<name> int64 = <value>
```

If the integer type is constrained to be nonnegative, than uint64 is used as the Go type for the constant.

REAL Value

A REAL type value is mapped to a Go floating point constant:

ASN.1 production:

```
<name> REAL ::= <value>
```

Generated code:

```
const Asn1v<name> float64 = <value>
```

ENUMERATED Value

The mapping of an ASN.1 enumerated value declaration to Go is to an integer constant.

ASN.1 production:

```
<name> <EnumType> ::= <value>
```

Generated code:

```
const <name> = <int value>
```

Where <int value> is the integer value associated with the ASN.1 enumerated value.

BIT STRING Value

ASN.1 values of type BIT STRING are mapped to Go variables of type asn1.BitString. The ASN.1 values may be defined using binary strings, hexadecimal strings, or using sets of named bits:

ASN.1 production:

```
<name> BIT STRING ::= '<bstring>'B
```

Generated code :

```
var Asn1v<name> asn1.BitString = asn1.BitString{ BitLength: <len>, Bytes: []byte{<data>
```

where <len> would be the length in bits and <data> would be a comma-separated list of hexadeciaml byte constants.

OCTET STRING Value

ASN.1 values of type OCTET STRING are mapped to Go variables of type []byte. The ASN.1 values may be defined using binary or hexadecimal strings. If a binary string is used and the bit count is not evenly divisible by eight, the last byte is zero-padded on the right with zero bits to form a full byte.

ASN.1 production:

```
<name> OCTET STRING ::= '<hstring>'H
```

Generated code :

```
var Asn1v<name> []byte = []byte{<data>}}
```

where <data> would be a comma-separated list of hexadeciaml byte constants.

Character String Value

All ASN.1 character string type values are mapped to Go string constants.

ASN.1 production:

```
<name> <string-type> ::= <value>
```

Generated code:

```
const Asn1v<name> string = <value>
```

Object Identifier Value Specification

ASN.1 values of type OCTET STRING are mapped to Go variables of type []uint64.

ASN.1 production:

```
<name> OBJECT IDENTIFIER ::= <value>
```

Generated code:

```
var Asn1v<name> []uint64 = []uint64{<arcs}>;
```

Where <arcs> is a comma-separated list of the OID integer arc values.

For example, consider the following declaration:

```
oid OBJECT IDENTIFIER ::= { ccitt b(5) 10 }
```

This would result in the following definition in the Go source file:

```
var Asn1vOid []uint64 = []uint64{ 0, 5, 10 }
}
```

Constructed Type Values

Generation of Go variables or constants for constructed type value specifications is not supported in this release.

Table Constraint Related Structures

The following sections describe changes to generated code that occur when the *-tables* or *-table-unions* option is specified on the command-line or when *Table Constraint Options* are selected from the GUI. This option causes additional code to be generated for items required to support table constraints as specified in the X.682 standard. This includes the generation of structures and classes for Information Object Classes, Information Objects, and Information Object Sets as specified in the X.681 standard.

Most of the additional items that are generated are read-only tables for use by the run-time for data validation purposes. However, generated structures for types that use table constraints are different than when table constraint code generation is not enabled. These differences will be pointed out.

For Go, the table constraint code generation model is roughly equivalent to C when the *-table-unions* option is used. That is, the generated structures are very similar to what would be generated for CHOICE constructs.

Unions Table Constraint Model

The unions table constraint model originated from common patterns used in a series of ASN.1 specifications in-use in 3rd Generation Partnership Project (3GPP) standards. These standards include Node Application Part B (NBAP) and S1AP and X2AP protocols in 4G network (LTE) standards. This pattern is repeated in 5G standards such as NGAP.

Generated Go Type Definitions for Message Types

The standard message type used by many specifications that employ table constraints is usually a SEQUENCE type with elements that use a relational table constraint that uses fixed-type and type fields. The general form is as follows:

```
   <Type> ::= SEQUENCE {
      <element1> <Class>.&<fixed-type-field> ({<ObjectSet>}),
      <element2> <Class>.&<fixed-type-field> ({<ObjectSet>)){@element1}
      <element3> <Class>.&<type-field> ({<ObjectSet>)){@element1}
}
```

In this definition, `<Class>` would be replaced with a reference to an Information Object Class, `<fixed-type-field>` would be a fixed-type field wtihin that class, and `<type-field>` would be a type field within the class. `<ObjectSet>` would be a reference to an Information Object Set which would define all of the possibilities for content within the message. The first element (<element1>) would be used as the index element in the object set relation.

An example of this pattern from the S1AP LTE specification is as follows:

```
   InitiatingMessage ::= SEQUENCE {
      procedureCode   S1AP-ELEMENTARY-PROCEDURE.&procedureCode
                         ({S1AP-ELEMENTARY-PROCEDURES}),
      criticality     S1AP-ELEMENTARY-PROCEDURE.&criticality
                         ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode}),
      value           S1AP-ELEMENTARY-PROCEDURE.&InitiatingMessage
                         ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode})
   }
```

In this definition, `procedureCode` and `criticality` are defined to be enumerated fixed types, and `value` is defined to be an open type field to hold variable content as defined in the object set definition.

The generated Go structure is similar to what is used to represent a CHOICE type. The structure consists of a choice selector value (T) followed by an inner structure (U) containing pointers to all of the alternative types that can appear in the field. This is the general form:

```
   type <Type> struct {
      <element1> <Element1Type>
      <element2> <Element2Type>

      /**
       * information object selector
       */
      T uint64

      /**
       * <ObjectSet> information objects
       */
      U struct {
         /**
          * <element1> : <object1-element1-value>
          * <element2> : <object1-element2-value>
          */
         <object1-name> *<object1-element3-type>

         /**
          * <element1> : <object2-element1-value>
          * <element2> : <object2-element2-value>
          */
         <object2-name>; *<object2-element3-type>

         ...
      }
   }
```

In this definition, the first two elements of the sequence would use the equivalent Go type as defined in the fixed-type field in the information object. The open type field (element3) would be expanded into the union structure as is shown. The T value would hold a sequential number which is generated to represent each of the choices in the referenced information object set. The union struct then contains an entry for each of the possible types as defined in the object set that can be used in the open type field. Comments are used to list the fixed-type fields corresponding to each open type field.

An example of the code that is generated from the S1AP sample ASN.1 snippet above is as follows:

```
const (
 S1APPDUDescriptionsS1APELEMENTARYPROCEDURESUNDEFTAG = 0
 S1APPDUDescriptionsS1APELEMENTARYPROCEDURESHandoverPreparationTAG = 1
 S1APPDUDescriptionsS1APELEMENTARYPROCEDURESHandoverResourceAllocationTAG = 2
 S1APPDUDescriptionsS1APELEMENTARYPROCEDURESPathSwitchRequestTAG = 3
        etc..
    )

    type InitiatingMessage struct {
        ProcedureCode uint64
        Criticality uint64
        Value struct {
            /**
             * information object selector
             */
            T uint64

           /**
             * S1AP-ELEMENTARY-PROCEDURES information objects
             */
            U struct {
                /**
                 * procedureCode: id-HandoverPreparation
                 * criticality: CriticalityReject
                 */
                // T = 1
                HandoverPreparation *HandoverRequired

                /**
                 * procedureCode: id-HandoverResourceAllocation
                 * criticality: CriticalityReject
                 */
                // T = 2
                HandoverResourceAllocation *HandoverRequest

                /**
                 * procedureCode: id-PathSwitchRequest
                 * criticality: CriticalityReject
                 */
                // T = 3
                PathSwitchRequest *PathSwitchRequest

                etc..

            }
        }
```

Note that the long names generated for the TAG constants can be reduced by using the <alias> configuration element.

Generated Go Type Definitions for Information Element (IE) Types

In addition to message types, another common pattern in 3GPP specifications is protocol information element (IE) types. The general form of these types is a list of information elements as follows:

```
<ProtocolIEsType> ::= <ProtocolIE-ContainerType> { <ObjectSet> }

<ProtocolIE-ContainerType> { <Class> : <ObjectSetParam> } ::=
   SEQUENCE (SIZE (<size>)) OF <ProtocolIE-FieldType> {{ObjectSetParam}}

<ProtocolIE-FieldType> { <Class> : <ObjectSetParam> } ::= SEQUENCE {
   <element1> <Class>.&<fixed-type-field> ({ObjectSetParam}),
   <element2> <Class>.&<fixed-type-field> ({ObjectSetParam}{@element1}),
   <element3> <Class>.&<Type-field> ({ObjectSetParam}{@element1})
}
```

There are a few different variations of this, but the overall pattern is similar in all cases. A parameterized type is used as a shorthand notation to pass an information object set into a container type. The container type holds a list of the IE fields. The structure of an IE field type is similar to a message type: the first element is used as an index element to the remaining elements. That is followed by one or more fixed type or variable type elements. In the case defined above, only a single fixed-type and variable type element is shown, but there may be more.

An example of this pattern from the S1AP LTE specification follows:

```
HandoverRequired ::= SEQUENCE {
   protocolIEs   ProtocolIE-Container   { { HandoverRequiredIEs} },
   ...
}

ProtocolIE-Container {S1AP-PROTOCOL-IES : IEsSetParam} ::=
      SEQUENCE (SIZE (0..maxProtocolIEs)) OF ProtocolIE-Field {{IEsSetParam}}

ProtocolIE-Field {S1AP-PROTOCOL-IES : IEsSetParam} ::= SEQUENCE {
      id            S1AP-PROTOCOL-IES.&id            ({IEsSetParam}),
      criticality   S1AP-PROTOCOL-IES.&criticality   ({IEsSetParam}{@id}),
      value         S1AP-PROTOCOL-IES.&Value         ({IEsSetParam}{@id})
}
```

In this case, standard parameterized type instantiation is used to create a type definition for the protocolIEs element. This results in the following element declaration in the HandoverRequired type:

```
ProtocolIEs []HandoverRequiredProtocolIEsElement
```

The type for the protocol IE list element is created in much the same way as the main message type was above:

```
type HandoverRequiredProtocolIEsElement struct {
    Id uint64
    Criticality uint64
    Value struct {
        /**
         * information object selector
         */
        T uint64

        /**
         * HandoverRequiredIEs information objects
         */
        U struct {
            /**
             * id: id-MME-UE-S1AP-ID
             * criticality: CriticalityReject
             * presence: PresenceMandatory
           */
            // T = 1
            HandoverRequiredIEsIdMMEUES1APID *uint64

            /**
             * id: id-eNB-UE-S1AP-ID
             * criticality: CriticalityReject
             * presence: PresenceMandatory
             */
```

```
                // T = 2
                HandoverRequiredIEsIdENBUES1APID *uint64

                ...
            }
        }
    }
```

In this case, the protocol IE id field and criticality are generated as usual using the fixed-type field type definitions. The open type field once again results in the generation of a union structure of all possible type fields that can be used.

Note in this case the field names are automatically generated (HandoverRequiredIEsIdMMEUES1APID, etc.). The reason for this was the use of inline information object definitions in the information object set as opposed to defined object definitions. This is a sample from that set:

```
   HandoverRequiredIEs S1AP-PROTOCOL-IES ::= {
      { ID id-MME-UE-S1AP-ID         CRITICALITY reject   TYPE MME-UE-S1AP-ID   PRESENCE m
      { ID id-HandoverType           CRITICALITY reject   TYPE HandoverType     PRESENCE m
   ...
```

In this case, the name is formed by combining the information object set name with the name of each key field within the set.

Chapter 7. Generated PER Functions

Generated PER Encode Functions

PER encode functions are generated when the *-per* (packed encoding rules), *-aper* (aligned packed encoding rules), or *-uper* (unaligned packed encoding rules), switch is specified on the command line. For Go, -per and -aper have the same meaning - aligned PER encoders will be generated. The -uper option is used to generate unaligned decoders. There is no option to switch between aligned and unaligned at run-time as there is for some other languages. For the remainder of this section, the acronym PER will be used to mean either aligned or unaligned PER.

For each ASN.1 production defined in the ASN.1 source file, a PER encode function is generated. This function will convert a populated Go variable of the given type into a PER encoded ASN.1 message.

Generated Go Function Format and Calling Parameters

The primary interface to encode data in Go is through the use of the generated *Marshal* function. This function has the same signature as that found in the Go encoding/asn1 package:

```
func Marshal(val interface{}) ([]byte, error)
```

The val argument holds a value of the type to be encoded. Note that the Marshal function only encodes types determined to be Protocol Data Unit (PDU) types. By default, a PDU type in a schema is a type not found to be referenced by any other type. This behavior can be overridden by using the *-pdu* command-line option to explicitly select types as PDU types.

The Marshal function returns the encoded message in memory as a byte slice. If any errors occur during encoding, the error result is returned in the error return value. The encoded value byte slice will be nil in this case.

Example of Calling the Go Marshal Function

The following code snippet was derived from the golang/sample_per/employee sample program in the ASN1C distribution:

```
package main

import (
 "employee/asn1gen"
 "encoding/hex"
 "fmt"
 "io/ioutil"
 "os"
)

func main() {
    var pdu asn1gen.PersonnelRecord

    // populate PDU type variable to be encoded
    fillName(&pdu.Name, "John", "P", "Smith")

    pdu.Title = "Director"
    pdu.Number = 51
    pdu.DateOfHire = "19710917"
    fillName(&pdu.NameOfSpouse, "Mary", "T", "Smith")

    // fill first child record

    var childInfo asn1gen.ChildInformation
    fillName(&childInfo.Name, "Ralph", "T", "Smith")
    childInfo.DateOfBirth = "19571111"
    pdu.Children = append(pdu.Children, childInfo)

    // fill second child record

    fillName(&childInfo.Name, "Susan", "B", "Jones")
    childInfo.DateOfBirth = "19590717"
    pdu.Children = append(pdu.Children, childInfo)

    // Marshal

    encodedData, err := asn1gen.Marshal(pdu)

    if err == nil {
        fmt.Printf("PersonnelRecord encoded successfully\n")
 fmt.Printf("%s\n", hex.Dump(encodedData))
 err = ioutil.WriteFile("message.per", encodedData, 0777)
    } else {
        fmt.Printf("Encoding failed: %s\n", err)
    }
```

How Marshal Works And Alternatives

The Marshal function is the most convenient way to encode, but taking a look at how it works also provides alternatives, should you want them.

First, we'll look at the other PER encode functions that are generated. There are two general forms:

```
   func PerEncode<type name>(pctxt *asn1rt.OSRTContext, value <go type>) (err error)

   func (pvalue *<type name>) PerEncode(pctxt *asn1rt.OSRTContext) (err error)
```

Here are two examples from the employee sample:

```
   func PerEncodeEmployeeNumber(pctxt *asn1rt.OSRTContext, value int64) (err error)

   func (pvalue *Name) PerEncode(pctxt *asn1rt.OSRTContext) (err error)
```

The first form is used when the Go type for the ASN.1 type is a built-in Go type - so basically it's used for the simple, nonconstructed ASN.1 types. The second form is used when the Go type is a generated type.

Now we can look at the Marshal function to see what it does and how these functions are used:

```
   // Create context object to manage encoding
   pctxt := new(asn1rt.OSRTContext)
   pctxt.InitEncode()   // Initialize for encoding
   pctxt.NewBitFieldList()   // Optional: enable bit tracing
   ...
   err = v.PerEncode(pctxt)   // Invoke the encode function; v is a generated type
   ...
   pctxt.FlushBuffer()
   pctxt.PrintBitFieldList("pdu")   // Optional: print bit trace

   return pctxt.BufferData(), err   // access the encoded data
```

Generated PER Decode Functions

PER decode functions are generated when the *-per* (packed encoding rules), *-aper* (aligned packed encoding rules), or *-uper* (unaligned packed encoding rules), switch is specified on the command line. For Go, -per and -aper have the same meaning - aligned PER decoders will be generated. The -uper option is used to generate unaligned encoders. There is no option to switch between aligned and unaligned at run-time as there is for some other languages. For the remainder of this section, the acronym PER will be used to mean either aligned or unaligned PER.

For each ASN.1 production defined in the ASN.1 source file, a PER decode function is generated. This function decodes an encoded ASN.1 message contained in a message buffer (byte slice) and stores the results in a data variable of a Go type corresponding to the ASN.1 production type.

Generated Go Function Format and Calling Parameters

The primary interface to decode data in Go is through the use of the generated *Unmarshal* function. This function has the same signature as that found in the Go encoding/asn1 package:

```
func Unmarshal(b []byte, val interface{}) (rest []byte, err error)
```

The b argument is used to pass a byte slice into the function containing the message to be decoded. The val argument holds a pointer to a variable of the type into which the message is to be decoded. Note that the Unmarshal function only supports decoding of types determined to be Protocol Data Unit (PDU) types. By default, a PDU type in a schema is a type not found to be referenced by any other type. This behavior can be overridden by using the *-pdu* command-line option to explicitly select types as PDU types>

The Unmarshal function returns any unused bytes after the message is decoded in the rest argument. This makes it possible to decode a buffer containing multiple messages by looping to decode a message and then assigning the rest result to the message buffer. When the buffer is empty, decoding would be terminated. If any errors occur during decoding, the error result is returned in the err error return value.

Example of Calling the Go Unmarshal Function

The following code snippet was derived from the golang/sample_per/employee sample program in the ASN1C distribution:

```
package main

import (
 "employee/asn1gen"
 "encoding/hex"
 "fmt"
 "io/ioutil"
 "os"
)

func main() {
    var pdu asn1gen.PersonnelRecord
    encoding, err := ioutil.ReadFile("message.per")
    if err != nil {
        fmt.Printf("Read message.per failed: %s\n", err)
 return
    }

    _, err = asn1gen.Unmarshal(encoding, &pdu)

    if err == nil {
        fmt.Printf("Decoding was successful:\n")
    } else {
        fmt.Printf("Decoding failed: %s\n", err)
    }
```

How Unmarshal Works And Alternatives

The Unmarshal function is the most convenient way to decode, but taking a look at how it works also provides alternatives, should you want them.

First, we'll look at the other PER decode functions that are generated. There are two general forms:

```
          func PerDecode<type name>(pctxt *asn1rt.OSRTContext) (value <go type>, err error

          func (pvalue *<type name>) PerDecode(pctxt *asn1rt.OSRTContext) (err error)
```

Here are two examples from the employee sample:

```
          func PerDecodeEmployeeNumber(pctxt *asn1rt.OSRTContext) (value int64, err error)

          func (pvalue *Name) PerDecode(pctxt *asn1rt.OSRTContext) (err error)
```

The first form is used when the Go type for the ASN.1 type is a built-in Go type - so basically it's used for the simple, nonconstructed ASN.1 types. The second form is used when the Go type is a generated type.

Now we can look at the Unmarshal function to see what it does and how these functions are used:

```
   // Create context object to manage encoding
   pctxt := new(asn1rt.OSRTContext)
   pctxt.InitDecodeBytes(b)   // Initialize for decoding; b holds the encoded data
   pctxt.NewBitFieldList()   // Optional: enable bit tracing
   ...
   err = v.PerDecode(pctxt)   // Invoke the decode function; v is a generated type
   ...
   pctxt.AlignBuffer()        // Optional: prepare to decode a second message
   pctxt.PrintBitFieldList("pdu")   // Optional: print bit trace
```

PER Bit Tracing

To print out a bit trace using ASN1C, invoke the appropriate functions after encoding or decoding a message and a marked-up binary dump of the PER encoded data is produced.

To enable bit tracing logic to be embedded in the generated code, the *-trace* option needs to be specified on the ASN1C command-line. If using the GUI, the *Add tracing diagnostic messages to code* box must be checked under the 'Constraints and Debugging' tab.

If enabled, a display similar to the following will be printed out from within the Marshal or Unmarshal function:

```
Dump of decoded bit fields:
employee childrenPresent
1xxxxxxx -------- -------- --------    80------    .---

employee.name.givenName length
-------- 00000100 -------- --------    --04----    -.--

employee.name.givenName data
-------- -------- 01001010 01101111    ----4a6f    --Jo
01101000 01101110 -------- --------    686e----    hn--
```

Here, a description can be seen of the field that was decoded, followed by 4 columns of a base 2 representation of the relevant input data, followed by a column with a hexadecimal representation of the same data, followed by a column with an ASCII representation of the same data. The hexadecimal and ASCII representations are only printed after all bits for a full byte have been processed, so these representations might not be printed on every line, and when they are printed, might correspond to binary bits on several preceeding lines.

The `x` values indicate padding bits used in aligned PER.

Chapter 8. Generated JSON Functions

The Go language contains built-in support for Marshaling/Unmarshaling data to and from JSON format. However, the default representation of the data is not always compliant with the JSON encoding rules (JER) as defined in ITU-T X.697.

Go provides mechanisms to customize the default marshalling behavior so that alternate JSON formats, including JER, can be supported. This is done through two code alterations:

- Setting custom JSON attributes
- Creating custom JSON Marshal/Unmarshal functions for specific types.

The following sections describe what is done in each of these areas and also general procedures for encoding and decoding JSON data.

Custom JSON Attributes

Custom JSON attributes are mainly used to alter the marshaling and unmarshaling of elements within a SEQUENCE or SET. They allow the following alterations to be performed:

- Change element name. This allows the ASN.1 name to be substituted for the Go element name.
- Omit empty items from the encoding (omitempty). This allows optional elements to be omitted.
- Remove elements (-). This allows elements to be removed that cannot be represented in JER (for example, an unknown extension).

Custom JSON Marshal/Unmarshal Functions

Custom JSON marshal and unmarshal functions are used to alter the encoded content for certain types. For example, the content of a BIT STRING is JER does not match what would be encoded by default. The JER encoding has a 'length' and 'value' member. A cusomized Marshal and Unmarshal method are therefore generated in the run-time to modify the encoded value to be in the JER format.

Invoking JSON Marshal/Unmarshal Functions

JSON marshal and unmarshal functions are invoked using the standard Go JSON calling conventions defined in encoding/json. For Marshal the format is:

```
func Marshal(v interface{}) ([]byte, error)
```

where v is the value to be encoded. The encode result is returned as a byte slice.

For Unmarshal, the format is:

```
func Unmarshal(data []byte, v interface{}) error
```

where data is a byte buffer containing the encoded JSON data to be unmarshaled (decoded) and v is the target variable into which the decoded data will be stored.

Chapter 9. Generated Print Functions

Generated Print Functions

The following options are available for generating code to print the contents of variables of generated types:

-print - This is the standard print option that causes print functions to be generated that output data to the standard output device (*stdout*).

-genPrtToStr - This option causes print functions to be generated that write their output to a string variable.

Print to Standard Output

The *-print* option causes functions to be generated that print the contents of variables of generated types to the standard output device. In this case, a function named *Print* is generated in the Marshal.go file with the following signature:

```
func Print(val interface{})
```

The val argument holds a value of the type to be printed. Note that the Print function only prints types determined to be Protocol Data Unit (PDU) types. By default, a PDU type in a schema is a type not found to be referenced by any other type. This behavior can be overridden by using the *-pdu* command-line option to explicitly select types as PDU types.

Print to String

The *-prttostr* option causes functions to be generated that print the contents of variables of generated types to a string variable. In this case, a function named *PrintToString* is generated in the Marshal.go file with the following signature:

```
func PrintToString(val interface{}) string
```

The val argument holds a value of the type to be printed. As was the case for the Print function, only variables of PDU types may be passed to the function for printing.

Chapter 10. IMPORT/EXPORT of Types

ASN1C allows productions to be shared between different modules through the ASN.1 IMPORT/EXPORT mechanism. The compiler parses but ignores the EXPORTS declaration within a module. As far as it is concerned, any type defined within a module is available for import by another module.

When ASN1C sees an IMPORT statement, it first checks its list of loaded modules to see if the module has already been loaded into memory. If not, it will attempt to find and parse another source file containing the module. The logic for locating the source file is as follows:

1. The configuration file (if specified) is checked for a <sourceFile> element containing the name of the source file for the module.

2. If this element is not present, the compiler looks for a file with the name <ModuleName>.asn where module name is the name of the module specified in the IMPORT statement.

In both cases, the –I command line option can be used to tell the compiler where to look for the files.

To avoid having ASN1C search for additional source files, simply provide the definition of each imported module in one of the source files specified as input to ASN1C. The imported module may be defined in its own source file, or multiple module definitions can be located in a single source file. Here is an example of combining modules into a single source file:

```
ModuleA DEFINITIONS ::= BEGIN
   IMPORTS B From ModuleB;

   A ::= B


END

ModuleB DEFINITIONS ::= BEGIN

   B ::= INTEGER
END
```

This entire fragment of code would be present in a single ASN.1 source file.