A Flexible Macro Processing Compiler for ASN.1

Paul V. Mellor, Jacqueline Balfour
Networks and Communications Laboratory
HP Laboratories Bristol
HPL-90-156
August, 1990

This document describes a new ASN.1 compiler that has several new and powerful features. It compiles full 1988 standard ASN.1 including all types and values, import/export, subtypes and so on. It handles recursive module definitions, and allows interactive correction of parse errors. It also has a macro processing capability, allowing macro definitions to be compiled on-the-fly, with the user supplying the semantics of the macro (over and above simple macro expansion). This capability also allows the production of user definable output. The compiler can be configured to produce output in any target language, for any function (not just simple encode/decode), and in any form that the user desires.
6.2 Processing Macro Definitions ............................. 13
6.3 Action Sections ........................................... 14
6.4 Annotation .................................................. 15

7 User Definable Output
7.1 Core Interface ............................................. 20

8 References .................................................... 22

List of Figures

1 Example Import Graph ................................. 7
2 Syntax Error ............................................... 8
3 Semantic Error ............................................ 9
4 Recursion Examples .................................. 12
1 Introduction

1.1 What is ASN.1

Abstract Syntax Notation One (ASN.1) was developed jointly by ISO [1, 2] and CCITT [3, 4], and it is now an International Standard ratified by both these organisations. Much of the importance of the ASN.1 language comes from the fact that it is the only internationally standardised language of its type, and it is used extensively in almost all ISO and CCITT communication based standards.

The purpose of the language is to allow the specification of data types and data values in an abstract (ie programming language independent) way. The ASN.1 specification can then be compiled automatically into routines that allow the transfer of this data between heterogeneous systems, without regard for how it is stored or represented on these systems.

1.2 Uses of ASN.1

Since its inception the uses of the language have diversified, primarily due to a facility called macros. This facility allows the syntax of the language to be extended on the fly. The semantics of these macros (ie what they actually do as opposed to what they look like) must be specified in English or some other means external to ASN.1 itself, and must then be implemented by users of the macros.

The standards bodies use macros quite extensively in order to extend the ASN.1 language. For example, the Remote Operations Standards [5] use a set of macros to define a Remote Procedure Call interface. If these macros are used then ASN.1 effectively becomes an interface definition language (or IDL).

In another standard, [6], a set of macros are used to give a specification for the structure of a complex system. The system is represented as a set of objects that interact in given ways. The macros can be used to define these interactions and also to recursively refine objects into sub-objects by treating them as systems in themselves. If these macros are used then ASN.1 effectively becomes a system specification language.

1.3 Why Write an ASN.1 Compiler

The role of ASN.1 is expanding past that of a simple data typing language, and it is becoming ever more important in the standards arena, and communication in general. Standards are constantly being produced with new specifications, and individual specifications are often interpreted differently by different sets of users, each with their own requirements. When working in this area it is imperative that an ASN.1 compiler be used that is extremely flexible and extensible, if constant and repeated redevelopment is to be avoided.
Also, the very purpose of ASN.1 is to allow communication between heterogeneous systems: that is, different machines, with different programming languages, and different compilers, and so on. Because of this it is only sensible to develop an ASN.1 compiler suited to such a heterogeneous environment, and which can produce multi-language output.

1.4 Compiler Features

The compilers available from other sources are fairly restrictive. This compiler stands out in that it has the following features:

- It supports full 1988 standard ASN.1, including subtypes, import/export, and all types and values. It does not change the syntax of normal ASN.1 in any way.

- It has full macro processing capability. This means that macro definitions are processed on-the-fly, and that instances can then be parsed and processed. Most compilers that handle macros do simple macro expansion of a fixed number of macros, by building the rules to do so into the compiler itself. This compiler can process any macro definition, and also allows semantics to be given to macros. For example, the ISO Network Management templates [7] have been written as a set of macros and annotated to produce C++ stub objects.

- It does not use any annotation of the ASN.1 source code (except in macro definitions). This means there is no need to modify the specifications appearing in standards, and that the source code is kept portable, and easy to maintain.

- It can compile recursive module definitions.

- It allows interactive correction of parse errors.

- An ASN.1 compiler normally supplies a concrete representation of the data types being compiled, along with a set of encode/decode routines. Since the compiler makes the decision as to what the concrete representation is, it is easy for it to generate the appropriate encode/decode routines. It can be seen that with this system the compiler fixes the concrete syntax for the programmer and not the other way round, as would be more desirable. We have fixed this problem by allowing the programmer to supply a specification of the concrete syntax required, in the form of CORE macros (see section 7). The compiler then produces the relevant output, which can be in any language, and in any form required by the application. Several standard sets of CORE macros have been developed, and users with special requirements can write their own or modify the ones supplied.
2 Overview

This section gives details of the overall operation of the compiler, so that the following sections can be more easily understood. Compilation proceeds in three passes, each of which is described below. These passes are passes of the internal tree, not of the source file. There is only a single pass of the source file (which means that errors can be corrected interactively). Note that each pass must succeed if the following pass is to be performed.

2.1 Parse Tree

When an ASN.1 source file is compiled a parse tree representation of that file is created in memory. This parse tree obviously has a structure which is very similar to the ASN.1 source syntax but not exactly the same.

Probably the major change is that none of the terminals are stored in this tree — there is simply no need for them. Also the compiler often flattens out structures (such as ‘COMPONENTS OF’), ties up defined types and values to their real definitions, resolves values, and so on. All these things are reflected in the parse tree, making its structure subtly different from the ASN.1 source structure.

2.2 Syntax Pass

This pass parses the source file and builds the parse tree. Only the syntactical structure of the source file is considered at this time, its correctness as an ASN.1 module is left for later. Thus, many parse tree elements are not filled in at this time, but are left for the next pass. However, macros are the exception to this rule since they are processed completely at this time in order to allow the parsing of instances that occur later in the same module.

Parsing occurs sequentially, that is, definitions are parsed in the order they appear in the source file.

It is during parsing that imported modules are read in and processed. Note that the useful definitions from the ASN.1 language have been implemented in a normal ASN.1 module called ‘Useful’. This module comes with the system and can be compiled as normal. It is automatically imported into every module (except the ‘Useful’ module itself) during compilation.

2.3 Semantic Pass

This pass runs through the parse tree built in the syntax pass, filling in all the missing information and checking that the ASN.1 is valid. This involves implementing all the restrictions given in text in the ASN.1 standard. This pass forms the bulk of the compiler. When the tree is complete it is written out to the modules export file.
In contrast to parsing, the semantic checks are performed in the order they appear in the parse tree (depth first). Each semantic check is performed only once, and of course imported types etc are not checked again since they have already been checked during compilation of the imported module.

2.4 Action Pass

This pass goes through the parse tree again performing all the required actions. All the output files are generated at this time. The rules to perform this pass are not part of the compiler itself, but are generated automatically by compiling the CORE macros (see section 7).

2.5 Ordering

The compiler is also prepared to change this sequence of events if it has to. This can result in types being checked at parse time, if they are needed in order to enable parsing of subsequent value definitions. It can also result in values being parsed at check time. This occurs when there is forward reference and it cannot be resolved when it is first encountered.

3 Import/Export

3.1 General

Import and export lists tie modules together into a (potentially very complex) directed graph. To complicate matters, this graph can be cyclic, meaning that modules can recursively import each other, or even import themselves. An example of such a set of imports is shown in figure 1.

In this example module A imports B and C; B imports D, E, and F; and so on. Note that module F is imported twice by two different paths. Note also that modules C, H and G form a cycle in the graph, since each imports the other. This example is referred to throughout the remainder of this section.

When a module imports another module the objects defined in the imported module may be needed by the compiler in order to compile the importing module. In order to facilitate this, export writes out the parse tree of a module, and import reads it in.

3.2 Full Imports

One method handling imports is to ensure that all the information necessary is available before the parse pass of the assignment list begins. This means that every module in the graph is read in. This is achieved by importing the first level
modules (B and C), and then recursively importing their imports (D, E, and F; and F, and G). This is repeated until all the necessary modules have been read.

This method suffers from the drawback that it often imports modules that are not needed. For example A may only use a single type defined in B, and this type may not require any of B's imports (i.e., it is defined fully in B itself). Nevertheless, even though D, E, and F are not used they will all be imported.

3.3 Lazy Imports

The method of import handling used in this compiler is lazy import. With this method only the first level imports are done before the parse pass of the assignment list begins (B, and C in the diagram). As the compilation proceeds the compiler will encounter references to objects in other modules. It then imports the modules in question, if it has not already done so, and searches them for the object required. When found, it is checked to see if it has references to other modules itself. And if so the process is repeated until the object is fully defined.

A list is kept of modules that have already been imported, and this is used to prevent re-import of modules which can be reached by more than one path. This method also breaks cycles in the graph.

4 Dealing With Errors

There are two general classes of error raised by the compiler, syntax errors and semantic errors. In general these are raised in the relevant passes, but syntax errors can occur in the semantic pass (due to late parsing of stored values), and semantic
Figure 2: Syntax Error

errors can also occur in the syntax pass (due to early checking of types needed to parse certain values).

Errors can also occur in the action pass, but these are rare and should never happen in normal use. The following section describes the error handling facilities in more detail.

4.1 Syntax Errors

Syntax errors are by their nature imprecise since the parser cannot tell exactly where the error occurred. These errors are reported by printing a series of symbols from the source file on the screen, and placing an error marker (‘<< here >>’) at the furthest point the parser reached before it failed.

For example, figure 2 shows a source file and the corresponding syntax error message.

Since the compiler cannot proceed past this point (due to synchronisation problems) this would normally terminate a compilation. However, having only one syntax error reported per compilation would make the development process tediously slow, and so a system has been developed to enable the interactive correction of parse errors as they occur.

The user is given the option of quitting the compilation or editing the source file. If editing is selected the compiler brings up an editor and allows the user to correct the source code. When editing is finished, compilation continues form the point previously reached.

It must be noted however that the compiler has already compiled everything up to the beginning of the list that contains the error marker, and so nothing in the source file before this point is allowed to change. The compiler detects this by keeping a
EG-4-2-0 DEFINITIONS ::= 
BEGIN 
A ::= SET {
    a INTEGER,
    b INTEGER, -- duplicate tag in set
    c BOOLEAN }
END 

error: duplicate tag [UNIV EXP 2]
error: incorrect element with identifier b
error: incorrect set type
error: incorrect assignment of A

Figure 3: Semantic Error

checksum, and so even comments and white space must be left unchanged. If the editing session does change the file before this point then the compiler detects this and produces an error. Any amount of text after the error marker can be changed as desired.

4.2 Semantic Errors

Semantic errors are by nature very precise. The compiler generates a hierarchical series of errors as it ascends back up the parse tree from the node that caused the error. By reading down this series of messages it is possible to precisely locate the point of failure.

For example, figure 3 shows a source file and the corresponding semantic error message.

The first error relates to the fact that the first problem the compiler detected was that the tag of a type had been duplicated. This caused it to fail the set element thus giving the second error message. Then, the ‘SET’ itself failed and an error was produced for this. The failure of this type caused it to fail the overall assignment for ‘A’ and produce the final error.

Note that assignment is a special case. If an assignment fails the whole module is not failed (immediately), the compiler carries on checking the rest of the assignments in that module. This means that semantic errors relating to many different assignments can be detected in a single pass.

Even though an assignment has failed it may still be used in other places in the
source code. Since it is incorrect, this may cause the compiler to generate other errors in these definitions. These will disappear automatically once the original assignment is corrected.

5 Recursive Module Sets

It is possible for ASN.1 definitions to be recursive. When recursion occurs inside a single module the compiler can deal with it since it has all the necessary information at its disposal. However, it is also possible for recursion to occur between modules, resulting in a cyclic set of import lists. This is more difficult for the compiler to resolve since it only sees a limited part of the source code at once. In order to solve this problem a system of cyclic compilation has been developed.

5.1 Cyclic Compilation

When cyclic compilation is specified (by setting the 'cyclic' option) the compiler will attempt to compile a file even though it cannot find all of its imported modules. It does this by regarding the error 'failed to find imported module' as a warning. Other errors related to import/export will still be errors and so will still cause compilation to fail.

Since not all the imported modules are present there will probably be a number of errors produced, but despite these the compiler will do its best to compile the file and will then export it. Normally, if errors occur no export file is produced, but with cyclic compilation an export file is produced despite these errors. However no action pass will be performed when cyclic is set, and thus no real output can be obtained from the erroneous file. The export file is marked with the number of elements contained in it which the compiler was unable to resolve (due to errors). If this number is zero then the file is considered fully resolved, and is equivalent to one compiled normally.

With this system each file in a recursion cycle can be compiled (despite the fact that its imported modules are not present). As each file is compiled an export file will be produced containing a number of unresolved elements. These export files will then be available for import by other files in the cycle. Since these modules will have more of their imported modules available the compilation should proceed with less errors (although it may still not be fully correct since some of the imported modules may still have unresolved symbols). The export files thus produced should therefore have less unresolved symbols than they had before.

When all of the files in the cycle have been compiled and have produced fully resolved export files, the cyclic option can be removed from the compiler and the files compiled as normal. This should produce a normal set of output files.

If the number of unresolved symbols stays static despite recompiations, then it is either because the files have real errors (ie independent of the fact that there is
a recursion cycle) and these should be fixed as normal; or it because there is an illegal recursive definition, as described below.

5.2 Recursion Correctness

Given that recursive modules can be compiled, it is still not certain that any recursion contained in the definitions themselves is acceptable. This section deals with this question, namely what forms of recursion exist and which of these are valid. Unfortunately the ASN.1 standard itself has little to say on this matter, and this is a defect we are attempting to correct.

Recursion can obviously only occur at places where the ASN.1 syntax itself is recursive, and even then recursion in the syntax does not necessarily mean that there will be cycles in the parse tree (for example 'type' can be recursive, but it is only through the use of 'defined.type' that actual cycles can occur).

The basic restriction imposed by the compiler is that in any recursion cycle there must be at least one type with a universal tag. This restriction ensures that the type is valid and that it is possible to supply values of this type. Note then that ANY, CHOICE, SELECTION, and MACRO types do not meet this requirement, and so cannot be used to make recursion cycles valid.

Some examples of valid and invalid recursion cycles are shown in figure 4.

6 Macro Support

Macros allow the syntax of the ASN.1 language to be extended. A macro definition consists of BNF like productions that define a grammar for instances of the macro. The semantics of these macro instances however are not specified. This is a considerable gap in the ASN.1 language that we have attempted to fill.

We have extended the macro syntax to allow annotation. This is a means by which the writer of the macro definition can specify a set of actions to take place when instances of the macro are encountered. In this way semantics can be given to the macro instances. In effect, we have converted the macro definition into a standard attribute grammar [8]. The actions that can be performed relate mainly to writing information to output files, doing some simple arithmetic and relation operations, and manipulating variables. However, combined with the expressive power of the macro productions themselves, this has been found to be sufficient.

When a macro definition is encountered it is translated by the compiler into a set of Prolog rules. Some of these rules parse instances of the macro (both type and value instances), other rules perform the relevant actions for these instances. Note that no rules are produced to check the semantics of a macro instance as there is no adequate way to specify this.

These rules are compiled automatically by the system and added into the compiler
6.1 Macro Definition Extensions

We have extended the syntax of macro definitions in several ways. These extensions are backward compatible with existing macro definitions.

6.1.1 Factorisation

Many macro definitions contain specifications for list like constructs, as in the following example:

\[
\text{List ::= Element | Element "," List}
\]

The rules generated directly from these productions can be inefficient, since repeated backtracking over 'Element' is involved. Hence we have chosen to introduce a factorisation facility. This allows common symbols of alternatives to be replaced by bracketed sections. All list constructs in macro definitions written using this factorisation method will be considerably quicker at parse time.

The previous example translates to:

\[
\text{List ::= Element | Element "," List}
\]
List ::= Element { ",” List | empty }

6.1.2 Macros within Macros

The use of macro instances within other macro definitions is a well known problem [9]. We allow this to occur with the condition that there is no forward reference. In order to parse an instance of a macro within the definition of another, the definition for the first macro must have already been processed. The one exception we allow to this rule is macros which reference themselves.

6.1.3 Macro References

We have extended the macro definition grammar to include 'macroreference' as a valid symbol. When a 'macroreference' appears in a macro definition it indicates that at this point in a macro instance an instance of the macro 'macroreference' should be encountered. In other words, it allows the nesting of macro TYPE NOTATION's.

6.2 Processing Macro Definitions

The macro parser is called by the compiler when a macro definition is encountered in the ASN.1 source file. The purpose of the macro parser is to process the definition and produce Prolog rules which can be incorporated into the compiler, allowing instances of the macro definition to be processed.

The initial task of the macro parser is to scan the macro definition, building up a macro parse tree which forms the basis for the remainder of the macro parse. Semantic checking is applied to this parse tree and another tree is produced, called the rule tree. This has the appropriate format and information to directly generate Prolog rules.

It is worth noting that any types and values appearing in macro definitions are not checked when the definition is processed. It is not possible to do this in general, since not all the information need necessarily be present. Consider the following macro production:

```
TagsOk ::= type(T1) type(T2) <Ok ::= SET {T1, T2}>
```

Since 'T1' and 'T2' are not available until an instance is parsed then it is impossible to tell, at definition time, whether the tags for the 'SET' are all unique (as they must be). Because of this, all macro checking is done on the instances, not the definitions, so it is possible to write a definition which, although passed by the compiler, will always produce incorrect instances. As in the following example:

```
TagsNotOk ::= type(T1) <NotOk ::= SET {T1, INTEGER, INTEGER}>
```
All the instances of this macro will fail as they all have two ‘SET’ elements with the same tag (ie two ‘INTEGERS’).

Two sets of Prolog rules are produced from a definition. One set (called parse rules) parse an instance of the macro and generate a parse tree for this instance. The second set (called action rules) traverse the tree produced by the parse rules collecting and formatting information as dictated by the user annotation.

Note that all this occurs within the syntax pass of the main compiler.

6.3 Action Sections

The syntax of macro definitions has been extended to allow the productions in it to be split into multiple sections. The first section contains only the productions that define the macro, as normal. The productions in this section cannot be annotated. Since this is the only mandatory section, backward compatibility is maintained. This section is known as the parse section since it is used to generate the parse rules, and determine the tree structure for the instances.

The remaining sections are known as action sections, each of which is named. The productions in these sections can include annotation and are used to generate the action rules that are applied to the instance tree to produce output. An unannotated action section results in action rules which traverse the tree but generate no output.

There can be one un-named action section (known as the default section), and the rules produces from this section are the rules used by default. Rules produces by other sections are called in a different way (see section 7.1).

An action section contains copies of those productions from the parse section that are deemed to be relevant (these are known as action productions). The user can also write any other productions felt necessary in order to implement the semantics of the macro, and these are known as extra productions. They effectively provide the user with a facility to write common subroutines that are useful in performing the actions required.

6.3.1 Tree Matching

All the action productions have implicit tree matching. This means that their structure describes the structure of the parse tree, and at run-time the relevant alternatives of a rule are selected based on the parse tree for the actual instance in question.

Extra productions do not have tree matching. This means that if alternatives are provided then the rule must contain a predicate for selecting which one of them will be executed (see section 6.4.6. In the absence of such a predicate the first option will be selected.
Extra productions can be called by action productions, but not vice versa. This is because action productions need to work on a specific part of the tree, and extra productions do not have access to the tree.

### 6.3.2 Pruning Actions

The action pass will do a complete depth-first pass of the parse tree. However, if no processing needs be done in a given branch of the tree then it is possible to stop the action pass visiting all these nodes. This is possible since any production from the parse section not requiring annotation (and not called by any other annotated production) may be omitted from the action section. No further processing will then occur in that branch. This holds for all productions, thus either the TYPE NOTATION or the VALUE NOTATION (but not both) may be omitted also.

Symbols of a production alternative which do not require annotation can be ignored by replacing the symbol by ‘Ignore’. This part of the tree will not be traversed during the action pass. The only symbol which cannot be ignored is ‘astring’ since this can cause ambiguity at run-time.

### 6.4 Annotation

By annotating a macro definition the user can effect rules which traverse the instance tree and process the information therein. The facilities for annotation are described in the following sections.

#### 6.4.1 Parameter Passing

Parameters facilitate the flow of information between the macro productions. Data can then be accumulated and transferred from one part of the instance tree to another. The syntax for macro definitions has been extended to allow parameters to be inserted after productions, macroreferences, TYPE and VALUE NOTATION’s, identifiers and numbers. A list of parameters is enclosed in round brackets and separated by commas. The valid annotation parameters are:

- Buffers (§6.4.3) which start with an uppercase letter. For example Buffer, Result, BUFF1.
- Variables (§6.4.2) which also start with an uppercase letter. For example X, Number, and REFERENCE.
- Constants, which are numeric or items starting with a lowercase letter. For example mask, flag, 1, and -256.
- Strings, enclosed by a pair of double quote characters. For example: "this is an example string"
The double quote character itself is represented by two double quotes.

- A dash, '-', in place of a variable name causes that parameter to be ignored.

6.4.2 Variables

Variables are not variables in the programming sense, but in the mathematical sense. That is they stand for a particular value, although that value may currently be unknown. This means that you cannot say ‘x = x + 1’ because it does not. From a computational viewpoint, variables can only be assigned once. Trying to re-assign them effectively acts as a predicate test. Consider, for example:

\[ X ::= 4 \] \[ X ::= 5 \]

The first predicate says ‘X is 4’. If X is currently unassigned then this succeeds and compilation can proceed. The second predicate says ‘X is 5’. This cannot succeed since X is 4, and so this alternative of the production will fail. If this is in an extra production, you will have supplied some other alternatives for it to try. If it's in a main production then the whole thing will fail.

A variable can be instantiated in three ways, by unification with a tree terminal, by arithmetic assignment (see section §6.4.5), and by predicate assignment (see section §6.4.6).

Unification with a tree terminal is implemented simply by giving the name of the variable as a parameter to one of the tree terminals ‘identifier’, ‘number’, ‘string’ and ‘astring’. The other terminal, ‘empty’, represents no data and therefore can have no parameter.

6.4.3 Buffers

Like variables, a buffer cannot be re-assigned. Buffers are initialised using the ‘$$’ construct. Within a pair of ‘$$’ the format and contents of a buffer are specified. The name of the buffer is given after the second ‘$$’. All whitespace is significant between a ‘$$’ pair. For example:

$$ All this text
"*((( !!!! will go into the buffer
$$ Buff1

As well as specifying text, the contents of another buffer may be included, by giving the name of that buffer inside single quotes. As in:

$$ Here -> 'Buff1' <- will be all the stuff from Buff1
$$ Buff2

16
6.4.4 Builtin Productions

There is a set of predefined productions which provide the user with some useful operations. These include ‘Output’ which writes buffers out to files, ‘Inform’ which generates error messages, and ‘Get-global’ and ‘Set-global’ which set and read global variables.

6.4.5 Arithmetic

Arithmetic and relational operations can be performed within a square bracket construct. The execution of the arithmetic expression will assign the resulting value to the variable. For example:

\[
\begin{align*}
[A &:= (B << 1) * 6 + 1] \\
[\text{Mask} &:= Z \text{ AND } 16]
\end{align*}
\]

6.4.6 Predicates

Predicates allow relational tests to be specified. Only one relational operation can appear in the brackets. If a conjunction is required then repeated predicates can be used.

\[
[X < 10] \ [Y = \text{yes}] \ [Z < 0] \ - \ \text{conjunction of three predicates}
\]

As well as having the effect of testing equality ‘=’ will cause an initialisation if it is compared to an uninitialised variable. Predicates can be used to effect a case like construct with alternatives of a extra production. The alternative with the relational construct evaluating to true is the one which is executed. The alternatives are attempted in order so that when one fails (ie the predicate fails) the next one is evaluated. Predicates must be used with care in action productions, since if because of the implicit tree matching, a failed predicate could cause the whole production to fail.

6.4.7 Annotation Example

The following example illustrates a macro definition with one action section (the default), that defines an array of integers.

\[
\begin{align*}
\text{Mod1} \ \text{DEFINITIONS} &::= \\
\text{MACRO INT-ARRAY} &::= \\
\text{BEGIN}
\end{align*}
\]
TYPE NOTATION ::= identifier Size-spec
VALUE NOTATION ::= empty
Size-spec ::= "[" number "]" | empty

ACTION

TYPE NOTATION(Result) ::= 
  identifier(Id) Size-spec(Size)
  $$\text{typedef int['Size'] 'Id';}\$$ Result
  Output(tmp,h,Result)

VALUE NOTATION ::= empty

Size-spec(Size) ::= 
  "[" number(Size) "]" |
  empty $$\text{MAX_ARRAY_SIZE}$ Size

END -- INT-ARRAY

END -- Mod1

Below are example instances and the code that results from the annotation applied to these instances. The output will be written to the file ‘Mod1.tmp.h’ since ‘Output’ uses the current module name and appends the first argument to this to get the file stem, in our case ‘tmp’. The second argument is the extension and the final argument the buffer to be output.

<table>
<thead>
<tr>
<th>Macro Instance</th>
<th>Generated Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X ::= INT-ARRAY</td>
<td>typedef int[10] fred;</td>
</tr>
<tr>
<td>fred[10]</td>
<td></td>
</tr>
<tr>
<td>Y ::= INT-ARRAY</td>
<td>typedef int[MAX_ARRAY_SIZE] bill;</td>
</tr>
<tr>
<td>bill</td>
<td></td>
</tr>
</tbody>
</table>

7 User Definable Output

Since there is already a method of specifying what form output will take, ie for macros, this same system has been extended to normal ASN.1. This has been done by specifying the ASN.1 syntax itself as a set of macros — one for BOOLEAN, one for INTEGER and so on.

As an example consider the following macro definition :-

CORE-REAL MACRO ::= 
BEGIN

18
Similar macros can be defined for every normal ASN.1 production — for each type and value, for import and exports, and even for macro definitions themselves! These macros are known as CORE macros. All CORE macros must be prefixed by 'CORE-' but this prefix is removed and not expected in the instance, or the tree.

In the example above note that the type notation is empty since the name of the macro must appear anyway. Also, note the reference to another CORE macro ‘CORE-SNDV’, which represents a signed-number. This macro would allow the definition of instances such as:

\[ R ::= \text{REAL} \]
\[ \text{rv1 } R ::= \{ 10, 4, 1645 \} \]
\[ \text{rv2 } R ::= \text{PLUS-INFINITY} \]

Unfortunately the parse rules obtained by compiling these definitions will not parse normal ASN.1, and so cannot be used as the parse pass of the compiler. There are many reasons for this, such as overlap of syntax and semantics, and the fact that the CORE macros represent the internal parse tree and not the ASN.1 source itself. Such parse rules would also be inefficient, and could not cope with the subtleties and ambiguities in the ASN.1 language. Because of this the parse rules for the compiler have been written by hand, and parse rules are not produced from CORE macros.

The action rules produced by compiling these definitions will match the normal ASN.1 parse tree, and perform actions on it. This then is the method used to get user definable output for all the normal ASN.1 constructs. Since the action rules must work on the parse tree, the CORE macros must reflect its structure, not the structure of the ASN.1 language itself. There are thus subtle differences in the CORE macros and the BNF definition of the ASN.1 language.

The action section of the REAL macro can be annotated to produce, for example, some ‘C’ definitions as follows:–
<table>
<thead>
<tr>
<th>Macro Instance</th>
<th>Generated Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R ::= \text{REAL} )</td>
<td>\text{typedef } R \text{ float;}</td>
</tr>
<tr>
<td>( \text{rv2 } R ::= {10, 5, -2} )</td>
<td>#define rv2 {5 * 10 ^ 2}</td>
</tr>
<tr>
<td>( \text{rv3 } R ::= \text{PLUS-INFINITY} )</td>
<td>#define rv3 PLUS_INFINITY</td>
</tr>
</tbody>
</table>

The user can specify the name of a file containing a set of compiled CORE macro rules that are loaded automatically on startup. Obviously several of these can be kept around, and switched between with ease in order to generate different kinds of output. For example one file could contain rules for generating ‘C’ output, one for ‘MODULA-2’ output, one for ‘C++’ and so on.

### 7.1 Core Interface

Almost all macro instances contain normal ASN.1 types and values, and so some mechanism needs to be provided to allow these to be incorporated into user annotation. This involves the user interfacing with the CORE macros.

The ‘type’ and ‘value’ symbols within macro definitions represent ASN.1 types and values within instances. The result of parsing these instances is a parse tree for a ‘type’ or a ‘value’, and this can be processed by the ‘CORE-TYPE’ and ‘CORE-VALUE’ macros.

In an action section, ‘type’ and ‘value’ can have a parameter which represents the parse tree for the actual value parsed. These parameters can be passed around as normal. Two builtin productions are supplied (called ‘Do-type’ and ‘Do-value’) which can use these parameters and access the action sections for CORE-TYPE and CORE-VALUE. If these productions are not referenced then no actions are performed for these symbols.

Both productions have the same parameter list. The first parameter is the name of the action section to be called. This may be a dash if the default section is required. The second parameter (which cannot be a dash) is a variable containing the tree returned by ‘type’ or ‘value’ in the action section. Any further parameters are dependent on the action section being referenced and will be specified by the writers of the particular CORE macro’s in use. They will generally contain the result that would normally be output when a type or value is encountered (such as a type definition, and encode and decode routines).

At run-time the existence of the section and the number of parameters given is checked. For example we have defined CORE-TYPE with two action sections

\[
\text{ACTION } -- \text{ default section} \\
\quad \text{TYPE NOTATION(SubtypeSize,Encode,Decode,Const)} ::= \ldots
\]

\[
\text{ACTION } \text{size } -- \text{ returns any subtype sizes} \\
\quad \text{TYPE NOTATION(Size)} ::= \ldots
\]
Calls to Do-type would look like:

```plaintext
-- get the Tree for type
type(Tree)

-- do the size action section
Do-type(size,Tree,Size)

-- do the default section and pass in
-- the value returned from the size section
Do-type(-,Tree,Size,Enc,Dec,Defs)
```

Note that 'Do-type' can also be used on a value parameter, in order to perform actions on the type of this value.

In some cases it is known that the actual 'type' or 'value' supplied in the instance will be a macro. There are two additional productions for these, 'Do-macro-type' and 'Do-macro-value'. These are used in the same way as 'Do-type' and 'Do-value' except the action rules executed are those for the corresponding macro. They therefore return the parameters as specified in the TYPE and VALUE NOTATION for the macro.
8 References


