6.8 For Further Reading

The Abstract Syntax Notation One is specified in [ISO87]. The corresponding CCITT recommendation is defined in [CCI88c].

Chapter 7

Pepsy

The *pepsy*(1) program is a table-driven replacement for *pepy* and *posy*. At present, all three compilers are available as a part of the standard distribution. In the next major release, *pepy* and *posy* will remain, but most likely will not be installed by default.

The remainder of this chapter describes how to use *pepsy*. By virtue of having presented the preceeding two chapters, only a terse discussion is needed!

7.1 Abstract Syntax Module

As with *posy*, the *pepsy* program reads a description of an *abstract syntax module* and produces the corresponding *C* structures along with several tables that define the mapping between the ASN.1 objects in the module and the *C* structures. These tables replace the work done by the *pepy* program.

Note that unlike *posy*, *pepsy* does not accept arbitary annotations in the abstract syntax module. It should be raw ASN.1 or the strictly limited augmentations of ASN.1 described later, either entered by hand, or produced by the *rosy* program. Note that you must tell *rosy* when *pepsy* will be used. This is accomplished with the '`-pepsy`' option, e.g.,

```
.ry.py:;        rosy -pepsy $(RYFLAGS) $<
```

7.2 PEPSY Environment

The *pepsy* program will produce two files after reading its abstract syntax module.

7.2.1 C Language Structures

For each type in the abstract syntax module, *pepy* will define a corresponding *C* language structure (`struct`), using the same rules as *posy* (see Section 5.2.1 on page 43).

7.2.2 Mapping tables

In addition to the definition of *C* language structures, *pepsy* will generate a *C* file containing several tables that define the mapping between the structures and the corresponding ASN.1 objects. As this is entirely an automatic process, the contents of the tables are relatively unimportant.

7.3 Known Deficiences

pepsy uses essentially the same front-end as the *posy*(1) program, so it has some limitations in the ASN.1 syntax it can accept. Consult Section 5.3 on page 52 for the details.

7.4 Running PEPSY

Here are a few things to know when running *pepsy*.

7.4.1 Options

The *pepsy* program has a few options to modify its behavior.

The '`-A`' switch directs *pepsy* to produce mapping tables for encoding, decoding, and printing.

The '`-a`' switch directs *pepsy* to augment the `#include` file it generates with commentary text.

The '`-d`' switch directs *pepsy*, to always ignore all *pepy*-style augmentations, including the "verbatim" actions occuring at the very beginning and end of the module.

The '`-f`' switch directs *pepsy* to generate *C* macros to deallocate the structures it defines (e.g., for type `type_MODULE_type`, a macro called `free_MODULE_type` is defined).

As with *posy*, the '`-h`' switch enables additional heuristics when *pepsy* generates a *C* language structure definition. See Section 5.4.1 on page 52. Note the '`-h2`' does not work for pepsy (yet).

Normally, *pepsy* prints the name of each type as it works. The '`-s`' switch disables this behavior.

7.4.2 Makefiles

By convention, input files to *pepsy* have the extension `.py`. In order to remain consistent with the naming scheme used by *posy*, rather than starting with file `module.py`, we assume we are starting with file `MODULE-asn.py`. Let us arbitrarily select the file named `MODULE-types.py` for the augmented abstract syntax module produced by *posy*. Hence

```
pepsy MODULE-asn.py
```

will produce:

`MODULE-types.h`: the *C* language structures; and,

`MODULE_tables.c`: the mapping tables.

Our *make*(1) rules are:

```
MODULE_tables.o:  MODULE_tables.c MODULE-types.h

MODULE_tables.c MODULE_tables.h: MODULE-asn.py
pepsy -A -f -h MODULE-asn.py
```

7.5 Pepy Enhancements to Pepsy

pepsy supports a limited subset of the *pepy* augmented ASN.1 grammar. This is for efficiency and to reduce the amount of work required to change from *pepy* to *pepsy*. See the file *pepsy/t3.py* for examples of these enhancements.

pepsy differs from pepy in one very important mannar. There is only one version of the ASN.1 grammar which contains all the augmentations for encoding and decoding. In contrast pepy has one definition for each task and consequently the two functions can function very differently. One common problem when turning *pepy* code into *pepsy* code is that the decoding and encoding functions used in the *pepy* version have calling conventions different to those required by *pepsy*. *pepsy* requires that the encoding function be passed a pointer to the C data type carrying the data and the decoding function be passed the *address* of the pointer. There are no such restrictions in *pepy* and often people use other methods, these need to be thoroughly checked and fixed up otherwise obscure core dumps are sure to occur.

pepsy also provides arbitary actions which can reference the encoded and unencoded data. Also provides the ability to slot in functions for handling particular types. Of *pepy*'s augmented ASN.1 the most efficient and encouraged are the value passing statements. This are compiled into very compact table entries and yield the best benefit of pepsy, its small size and corresponding compile times.

7.5.1 Verbatim Actions

Verbatim actions are supported exactly as per *pepy*. The verbatim actions are placed in the `MODULE\_tables.c`, at the start or the end as appropriate. This allows files to be conviently included that contain definitions for types used later in the augmented grammar. It is also convient place for functions that are only relevant to the encoding or decoding of that modules data structures.

7.5.2 Value Passing Statements

These are a restriction of the *pepy* grammar for value passing statements. The C expressions are limited to specifying the field of the C structure carrying data corresponding to that typereference. See Figure 7.1 for example.

Here the C data type carrying the data from which the data is encoded from or decoded into is specified by the

```
[[ P struct pepy_refs * ]]
```

```
Pepy-refs [[ P struct pepy_refs * ]] ::= SEQUENCE {
    [1] IMPLICIT INTEGER [[ i parm->t_int ]],
    [2] IMPLICIT ENUMERATED [[ i parm->t_enum ]] {
        orange(1), apple(2), pear(3), bannana(4) },

    [3] IMPLICIT OCTET STRING [[ q parm->t_qbuf ]],
    [4] IMPLICIT OCTET STRING [[ s parm->t_string ]],
    [5] IMPLICIT OCTET STRING [[ o parm->t_ostring $ parm->t_olen ]],
    [6] IMPLICIT BIT STRING [[ t parm->t_pe ]],                          10
    [7] IMPLICIT BIT STRING [[ x parm->t_bstring $ t_blen ]],
    [8] IMPLICIT BOOLEAN [[ b parm->t_bool ]],
    [9] IMPLICIT OBJECT IDENTIFIER [[ O parm->t_oid ]],
    [10] IMPLICIT REAL [[ r parm->t_real ]],
    [11] ANY [[ a parm->t_any ]],
    [12] Def-types [[ p parm->t_def ]],
    [13] Opt-types [[ p parm->t_opt ]] OPTIONAL

    }
```

Figure 7.1: Pepsy value passing

augmentation. This is very useful as it allows the programmer to design their own data structures instead of being forced to use those produced by *pepsy*. This is especially useful for people changing over from another encoder/decoding system which uses different data structures to that of *posy* or *pepsy*. It is still restrictive in that integers must be passed in ints and booleans must be passed in chars but is considerable more flexible than *posy*.

All the value passing types are supported with some being interpreted slightly differently and some news ones. The usage is the same as pepy except, as mentioned before, the C expressions are limited to structure fields.

The name of the pointer to the structure is not used at all and in fact the C expression can be abreviated to just the name of the field which contains the data. Note if the name of a C type is not specified for the typereference pepsy assumes the C type it generates for the typereference. It always generates the definitions for these typereferences in the *-types.h file regardless of whether another type is specified by a `P` value passing statement in the grammar.

- `[[ t field]]` — a bit vector. This is used for BIT STRING types and takes a bit vector as described in Section 5.4.5 in *Volume One*. Note that a bitvector is *not* the same as a Presentation Element and failing to convert between them properly will give inconsistant results.

- `[[p field]]` — a pointer type. value passing statement is interpreted slightly differently to its pepy usage. This specifies the field which should be treated as a pointer to another C type which carries the information corrresponding to that ASN.1 typereference. The structure containing the data can be the current structure, that is to say the data is contained in fields of the current structure and so it shouldn't follow any pointer to it. This is specified by putting a `*` in the field arguement.

- `[[T type $ field]]` — a compact way of specifying a `P` and a `p` value passing statements. This is essential for contructed types like sequence and set which need both value passing statements but the normal (long) form is not suitable. If the Information in a contructed form is carried in a different type, perhaps a substructure, there is no way to specify this new type and how to get to it from the existing data structure.

- `[[E fn-name]]` — Encoding function

- `[[D fn-name]]` — Decoding function These entries give the name of a function which encodes/decodes that particular type. This allows people to get right in if they want and hand code routines for encoding/decoding particular types. Perhaps for efficiency or convience when converting from *pepy*. As *pepy* generates functions these could be suitable modified to conform with the convention outlined below and just included in a verbatim action. This may allow a particularly tricky piece of pepy code to be conviently carried over. Obviously if this is used every where the *pepsy* benefit of order of magnitude reduction in code size will be lost, so it is worth avoiding if possible.

Note that the type *must* match that of the ASN.1 being encoded/decoded. This means that the ASN.1 real will expect a double as it field and a boolean expects a char. Anything different will behave in a machine dependant mannar!

7.5.3 Constructed Types - Sequence and Set

As described above these ASN.1 types need to have a `T` value passing statement when used inside a type reference if they are not using the *pepsy* generated C data structure. See Figure 7.2 for example. The outer most sequence or set does not need them of course. Notice in the example that all the `T` fields are using the parent data structure for their information. This illustrates that `*` and `*parm` are considered equivalent, later case is supported for compatility with the augmented ASN.1 grammar produced by *posy*.

Figure 7.3 shows an example with a different C type used for the SEQUENCE OF and SET OF to that specified in the `P` value statement. In the case of the repeating types it is necessary to have a link list to carry the information, with a *next* pointer a part of the structure. The *next* field is indicated by placing it inside the `<<` and `>>` backets as shown. If this control statement is left out it will assume there is a field called "next" in the current structure which does the job. This is consistent with what *posy* generates. Note in the component label `b` the repeated type is a defined type and its data must be carried in the `rep_elem` structure as the encoding/decoding routine is called with the current (`rep_elem`) structure.

```
   Single-type [[ P struct pepy_refs1 * ]] ::= SEQUENCE {
      a SEQUENCE [[ T struct pepy_refs1 * $ * ]] {
            [1] IMPLICIT INTEGER [[ i parm->t_int ]] OPTIONAL << opt_set $ O
PT_INT1 >>
         },
      b SEQUENCE [[ T struct pepy_refs1 * $ * ]] {
            [2] INTEGER [[ i parm->t_int1 ]] OPTIONAL << opt_set $ OPT_INT2
>>
         },
                                                                        10
      c SET [[ T struct pepy_refs1 * $ *parm ]] {
         [3] IMPLICIT ENUMERATED [[ i parm->t_enum ]] {
orange(1), apple(2), pear(3), bannana(4) } OPTIONAL << opt_set $ OPT_ENUM1 >>
      },
      d SET [[ T struct pepy_refs1 *  $ *parm ]] {
         [4] ENUMERATED [[ i parm->t_enum1 ]] {
orange(1), apple(2), pear(3), bannana(4) } OPTIONAL << opt_set $ OPT_ENUM2 >>
      }
   }
```

Figure 7.2: Constructed Type Example

```
Repeated [[ P struct repeats * ]]::= SEQUENCE {
    a SEQUENCE OF [[ T struct rep_int * $ rp_sq1 ]]
        <<r>> [1] INTEGER [[ i i ]],

    b SEQUENCE OF [[ T struct rep_elem * $ rp_sq2 ]] <<r_next>>
            Rep-elem [[ p * ]],

    c SET OF [[ T struct rep_int * $ rp_st1 ]] <<r>> INTEGER [[ i i ]],

    d SET OF [[ T struct rep_elem * $ rp_st2 ]] <<r_next>>                10
            [3] IMPLICIT Rep-elem [[ p * ]],

    e CHOICE [[ T struct repeats  * $ * ]] <<rp_choice>> {
            [1] INTEGER [[ i rp_int ]],
            [2] BOOLEAN [[ b rp_bool ]],
            [3] OCTET STRING [[ o rp_ostring $ rp_int ]]
        }
    }
```

Figure 7.3: Repeated Type Example